
MICROSOFT®
The High Performance Software™

Microsoft® C Compiler

Language Reference

Microsoft Corporation

If you have comments about the software, complete the Software Problem Report at the back of this manual and return it to Microsoft Corporation.

If you have comments about the software documentation, complete the Documentation Feedback reply card at the back of this manual and return it to Microsoft Corporation.

Document Number 410840018-400-R01-0486

Contents

Tables

Chapter 1
Introduction

1.1 Overview

The C language is a general-purpose programming language well known for its efficiency, economy, and portability. While these advantages make it a good choice for almost any kind of programming, C has proved to be especially useful in systems programming because it allows programmers to write fast and compact programs and to transport those programs to other systems. In many cases, well-written C programs are comparable in speed to assembly-language programs, and they offer the advantages of easier maintenance and greater readability.

C combines efficiency and power in a relatively small language. C does not include built-in functions to perform tasks such as input and output, storage allocation, screen manipulation, and process control. Instead, C programmers rely on run-time libraries to perform such tasks.

This design contributes to C's adaptability and compactness. Because the language is relatively confined, it does not assume or impose a particular programming model. Run-time routines provide support as needed, allowing the programmer to minimize their use, if desired, or to tailor run-time routines for special purposes.

The design also helps to isolate language features from processor-specific features in a particular C implementation, thus aiding programmers who want to write portable code. The strict definition of the language makes it independent of any particular operating system or machine; at the same time, programmers can easily add system-specific routines to take advantage of a particular machine's efficiencies.

Some of the significant features of the C language are as follows:

- C provides a full set of loop, conditional, and transfer statements to control program flow logically and efficiently and to encourage structured programming.
- C offers an unusually large set of operators. Many of C's operators correspond to common machine instructions, allowing a direct translation into machine code. The variety of operators lets the programmer specify different kinds of operations clearly and with a minimum of code.
- C's data types include several sizes of integers, as well as single- and double-precision floating-point types. The programmer can design more complex data types, such as arrays and data structures, to suit specific program needs.

- C programmers can declare "pointers" to variables and functions. A pointer to an item corresponds to the machine address of that item. Using pointers wisely can increase program efficiency considerably, since pointers let the programmer refer to items in the same way the machine does. C also supports pointer arithmetic, allowing the programmer both to access and manipulate memory addresses directly.
- The C preprocessor, a text processor, acts on the text of files before compilation. Among its most useful applications for C programs are the definition of program constants, the substitution of function calls with faster macro look-alikes, and conditional compilation. The preprocessor is not limited to processing C files; it can be used on any text file.
- C is a flexible language, leaving much of the decision making up to the programmer. In keeping with this attitude, C imposes few restrictions in matters such as type conversion. While this is often an asset, C programmers must know the language well to understand how their programs will behave.

1.2 About This Manual

The *Microsoft® C Compiler Language Reference* defines the C language as implemented by Microsoft Corporation. It is intended as a reference for programmers who have experience in C or in another programming language. Knowledge of programming fundamentals is assumed.

Note

If you want a quick overview of how Microsoft C compares with the definition of C found in Appendix A of *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie, turn to Appendix A of this manual. Appendix B of this manual summarizes the syntax of the C language as defined by Microsoft.

The run-time library functions available for use in Microsoft C programs are discussed in a separate library reference manual.

Consult your system documentation for an explanation of how to compile and link C programs on your system. Your system documentation also contains information specific to the implementation of C on your system.

This manual is organized as follows:

Chapter 2, "Elements of C," describes the letters, numbers, and symbols that can be used in C programs and the combinations of characters that have special meanings to the C compiler.

Chapter 3, "Program Structure," discusses the components and structure of C programs and explains how C source files are organized.

Chapter 4, "Declarations," describes how to specify the attributes of C variables, functions, and user-defined types. C provides a number of predefined data types and allows the programmer to declare aggregate types and pointers.

Chapter 5, "Expressions and Assignments," describes the operands and operators that form C expressions and assignments. The chapter also discusses the type conversions and side effects that may accompany the evaluation of expressions.

Chapter 6, "Statements," describes C statements. Statements control the flow of program execution.

Chapter 7, "Functions," discusses features of C functions. In particular, this chapter explains how to define, declare, and call a function and describes function parameters and return values.

Chapter 8, "Preprocessor Directives and Pragmas," describes the instructions recognized by the C preprocessor. The C preprocessor is a text processor automatically invoked before compilation. This chapter also introduces pragmas, which are instructions to the compiler that are placed in the source file.

Appendix A, "Differences," lists the differences between Microsoft C and the description of the C language found in Appendix A of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

Appendix B, "Syntax Summary," summarizes the syntax of the C language as implemented by Microsoft.

The remainder of this chapter describes the notational conventions used throughout the manual.

1.3 Notational Conventions

This manual uses the following notational conventions:

Convention	Meaning
Bold	Bold type indicates text that must be typed exactly as shown. Text that is shown in bold type includes C keywords, such as **goto** and **char**, and operators, such as the addition operator (+) and the multiplication operator (*).
Italics	Italicized terms mark the places in syntax specifications and in the text where specific terms appear in an actual C program. For example, in **goto** *name*; *name* is italicized to show that this is a general form for the **goto** statement. In an actual program statement, the user supplies a particular identifier for the placeholder *name*. Occasionally, italics are used to emphasize particular words in the text.

`Examples`	Examples of C programs and program elements appear in a special typeface to look similar to listings on the screen or the output of commonly used computer printers:
	`int x, y;`
	`.`
	`.`
	`.`
	`swap (&x, &y);`

Ellipsis dots

Ellipsis dots may be vertical or horizontal. In the following example, the vertical ellipsis dots indicate that zero or more declarations, followed by one or more statements, may appear between the braces:

{
[[*declaration*]]
.
.
.
statement
[[*statement*]]
.
.
.
}

Vertical ellipsis dots are also used in program examples to indicate that a portion of the program has been omitted. For instance, in the following excerpt, two program lines are shown. The ellipsis dots between the lines indicate that intervening program lines occur but are not shown:

```
int x, y;
.
.
.
swap (&x, &y);
```

Horizontal ellipsis dots following an item indicate that more items having the same form may appear. For instance,

= { *expression* [[, *expression*]]...}

indicates that one or more expressions separated by commas may appear between the braces ({ }).

[[Double brackets]]

Double brackets enclose optional items in syntax specifications. For example,

return [[*expression*]]**;**

is a syntax specification showing that *expression* is an optional item in the **return** statement.

"Quotation marks"	Quotation marks set off terms defined in the text. For example, the term "token" appears in quotation marks when it is defined. Some C constructs, such as strings, require quotation marks. Quotation marks required by the language have the form "" rather than " ". For example, `"abc"` is a C string.
SMALL CAPITALS	Names of special key combinations, such as CONTROL-Z, appear in small capital letters.

Chapter 2
Elements of C

2.1 Introduction

This chapter describes the elements of the C programming language. The elements of the language are the names, numbers, and characters used to construct a C program. In particular, this chapter describes the following:

- Character sets
- Constants
- Identifiers
- Keywords
- Comments
- Tokens

2.2 Character Sets

Two character sets are defined for use in C programs: the C character set and the representable character set. The C character set consists of the letters, digits, and punctuation marks that have a specific meaning to the C compiler. C programs are constructed by combining the characters of the C character set into meaningful statements.

The C character set is a subset of the representable character set. The representable character set consists of all letters, digits, and symbols that a user can represent graphically with a single character. The extent of the representable character set depends on the type of terminal, console, or character device being used.

A C program can contain only characters from the C character set, with the exceptions of string literals, character constants, and comments, which can use any representable character. Each character in the C character set has an explicit meaning to the C compiler. The compiler generates error messages when it encounters misused characters or characters not belonging to the C character set.

The following sections describe the characters and symbols of the C character set and explain how and when to use them.

2.2.1 Letters and Digits

The C character set includes the uppercase and lowercase letters of the English alphabet and the 10 decimal digits of the Arabic number system:

- Uppercase English letters

  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- Lowercase English letters

  a b c d e f g h i j k l m n o p q r s t u v w x y z
- Decimal digits

  0 1 2 3 4 5 6 7 8 9

These letters and digits can be used to form the constants, identifiers, and keywords described later in this chapter.

The C compiler treats uppercase and lowercase letters as distinct characters. If a lowercase a is specified in a given item, you cannot substitute an uppercase A in its place; you must use the lowercase letter.

2.2.2 White-Space Characters

Space, tab, line-feed, carriage-return, form-feed, vertical-tab, and new-line characters are called white-space characters because they serve the same purpose as the spaces between words and lines on a printed page. These characters separate user-defined items, such as constants and identifiers, from other items within a program.

A CONTROL-Z character is treated as an end-of-file indicator. The compiler disregards any text following the CONTROL-Z mark.

The C compiler ignores white-space characters unless they are used as separators or as components of character constants or string literals. This means you can use extra white-space characters to make a program more readable. Comments are also treated as white space (see Section 2.6).

2.2.3 Punctuation and Special Characters

The punctuation and special characters in the C character set are used for a variety of purposes, from organizing the text of a program to defining the tasks to be carried out by the compiler or by the compiled program. Table 2.1 lists these characters.

Table 2.1

Punctuation and Special Characters

Character	Name	Character	Name
,	Comma	!	Exclamation mark
.	Period	¦	Vertical bar
;	Semicolon	/	Forward slash
:	Colon	\	Backslash
?	Question mark	~	Tilde
'	Single quotation	_	Underscore
"	Double quotation	#	Number sign
(	Left parenthesis	%	Percent sign
)	Right parenthesis	&	Ampersand
[	Left bracket	^	Caret
]	Right bracket	*	Asterisk
{	Left brace	-	Minus sign
}	Right brace	=	Equal sign
<	Left angle bracket	+	Plus sign
>	Right angle bracket		

These characters have special meaning to the C compiler; their use in the C language is described throughout this manual. Punctuation characters in the representable character set that do not appear in this list can be used only in string literals, character constants, and comments.

2.2.4 Escape Sequences

Escape sequences are special character combinations that represent white-space and nongraphic characters in strings and character constants. They are typically used to specify actions such as carriage returns and tab movements on terminals and printers and to provide literal representations of characters that normally have special meanings, such as the double quote (") character. An escape sequence consists of a backslash followed by a letter or combination of digits. Table 2.2 lists the C language escape sequences.

Table 2.2
Escape Sequences

Escape Sequence	Name
\n	New line
\t	Horizontal tab
\v	Vertical tab
\b	Backspace
\r	Carriage return
\f	Form feed
\a	Bell (alert)
\'	Single quote
\"	Double quote
\\	Backslash
\ *ddd*	ASCII character in octal notation
\x *dd*	ASCII character in hexadecimal notation

If the backslash precedes a character not included in the list above, the backslash is ignored and that character is represented literally. For example the pattern \c represents the character c in a string literal or character constant.

The sequences \ *ddd* and \x *dd* allow any character in the ASCII (American Standard Code for Information Interchange) character set to be given as a three-digit octal or a two-digit hexadecimal character code. For example, the backspace character can be given as \010 or \x08. The ASCII null character can be given as \0 or \x0.

Only the digits 0 through 7 can appear in an octal escape sequence, and at least one digit must appear. However, fewer than three digits can be specified. For example, the backspace character can be given in octal notation as \10. Similarly, a hexadecimal escape sequence must contain at least one digit, but the second digit can be omitted. Thus, the hexadecimal escape sequence for the backspace character can be given either as \x08 or as \x8.

Note

When using octal and hexadecimal escape sequences in strings, it is safest to give all digits of the escape sequence (three digits for octal and two digits for hexadecimal escape sequences). Otherwise, if the character immediately following the escape sequence happens to be an octal or hexadecimal digit, it is interpreted as part of the sequence. For example, if the string `\x7Bell` were printed, the result would be `{ell` because `\x7B` is interpreted as the ASCII left brace character (`{`). The string `\x07Bell` (note the `0`) is the correct way to represent the bell character followed by the word `Bell`.

Escape sequences allow nongraphic control characters to be sent to a display device. For example, the escape character, `\033`, is often used as the first character of a control command for a terminal or printer.

Nongraphic characters should always be represented by escape sequences because using a nongraphic character in a C program has unpredictable results.

The backslash character (\) used to introduce escape sequences also functions as a continuation character in strings and in preprocessor definitions. When a new-line character follows the backslash, the new line is disregarded, and the next line is treated as part of the previous line.

2.2.5 Operators

Operators are special character combinations that specify how values are to be transformed and assigned. The compiler interprets each of these character combinations as a single unit, called a "token" (see Section 2.7).

Table 2.3 lists the characters that form C operators and gives the name of each operator. Operators must be specified exactly as they appear in the tables, with no white space between the characters of multicharacter operators. The **sizeof** operator is not included in this table; it consists of a keyword (**sizeof**) rather than a symbol.

Table 2.3
Operators

Operator	Name
!	Logical NOT
~	Bitwise complement
+	Addition
–	Subtraction, arithmetic negation
*	Multiplication, indirection
/	Division
%	Remainder
<<	Left shift
>>	Right shift
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equality
!=	Inequality
&	Bitwise AND, address of
|	Bitwise inclusive OR
^	Bitwise exclusive OR
&&	Logical AND
||	Logical OR
,	Sequential evaluation
?:	Conditional[a]
++	Increment
--	Decrement
=	Simple assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Remainder assignment
>>=	Right-shift assignment

Table 2.3 *(continued)*

Operator	Name
<<=	Left-shift assignment
&=	Bitwise AND assignment
|=	Bitwise inclusive OR assignment
^=	Bitwise exclusive OR assignment

[a] The conditional operator is a ternary operator, not a multicharacter operator. The form of a conditional expression is the following: *expression* ? *expression* : *expression*

For a complete description of each operator, see Chapter 5, "Expressions and Assignments."

2.3 Constants

A constant is a number, a character, or a string of characters that can be used as a value in a program. The value of a constant does not change from execution to execution.

The C language has four kinds of constants: integer constants, floating-point constants, character constants, and string literals. The following sections define the format and use of each.

2.3.1 Integer Constants

An integer constant is a decimal, octal, or hexadecimal number that represents an integer value. A decimal constant has the form

digits

where *digits* is one or more decimal digits (0 through 9).

An octal constant has the form

0*odigits*

where *odigits* is one or more octal digits (0 through 7). The leading 0 is required.

A hexadecimal constant has one of the following forms:

0x*hdigits*
0X*hdigits*

where *hdigits* is one or more hexadecimal digits (0 through 9 and either uppercase or lowercase "a" through "f"). The leading 0 is required and must be followed by **x** or **X**.

No white-space characters can appear between the digits of an integer constant. Table 2.4 illustrates the form of integer constants.

Table 2.4

Examples of Integer Constants

Decimal Constants	Octal Constants	Hexadecimal Constants
10	012	0xa or 0xA
132	0204	0x84
32179	076663	0x7dB3 or 0x7DB3

Integer constants always specify positive values. If negative values are required, the minus sign (–) can be placed in front of the constant to form a constant expression with a negative value. The minus sign is treated as an arithmetic operator.

Every integer constant is given a type based on its value. A constant's type determines what conversions must be performed when the constant is used in an expression or when the minus sign (–) is applied. Decimal constants are considered signed quantities and are given **int** type, or **long** type if the size of the value requires it.

Octal and hexadecimal constants are given **int**, **unsigned int**, **long**, or **unsigned long** type depending on the size of the constant. If the constant can be represented as an **int**, it is given **int** type. If it is larger than the maximum positive value that can be represented by an **int**, but small enough to be represented in the same number of bits as an **int**, it is given **unsigned int** type. Similarly, a constant that is too large to be represented as an **unsigned int** is given **long** type, or **unsigned long** type, if necessary.

Table 2.5 shows the ranges of values and the corresponding types for octal and hexadecimal constants on a machine where the **int** type is 16 bits long.

Table 2.5

Types Assigned to Octal and Hexadecimal Constants

Hexadecimal Range	Octal Range	Type
0x0 – 0x7FFF	0 – 077777	int
0x8000 – 0xFFFF	0100000 – 0177777	unsigned int
0x10000 – 0x7FFFFFFF	0200000 – 017777777777	long
0x80000000 – 0xFFFFFFFF	020000000000 – 030000000000	unsigned long

The consequence of the above typing rules is that hexadecimal and octal constants are not sign-extended when they are converted to longer types. (For a discussion of type conversions, see Chapter 5, "Expressions and Assignments.")

The programmer can force any integer constant to be given **long** type by appending the letter "l" or "L" to the end of the constant. Table 2.6 illustrates **long** integer constants.

Table 2.6

Examples of Long Integer Constants

Decimal Constants	Octal Constants	Hexadecimal Constants
10L	012L	0xaL or 0xAL
791	01151	0x4f1 or 0x4F1

Types are described in Chapter 4, "Declarations," and conversions are described in Chapter 5, "Expressions and Assignments."

2.3.2 Floating-Point Constants

A floating-point constant is a decimal number representing a signed real number. The value of a signed real number includes an integer portion, a fractional portion, and an exponent. Floating-point constants have the form

[[*digits*]][.*digits*][[**E**[–]]*digits*]]

where *digits* is one or more decimal digits (0 through 9), and **E** (or **e**) is the exponent symbol. Either the digits before the decimal point (the integer portion of the value) or the digits after the decimal point (the fractional portion) can be omitted, but not both. The decimal point can be omitted only when an exponent is given.

The exponent consists of the exponent symbol followed by a possibly negative constant integer value. No white-space characters can separate the digits or characters of the constant.

Floating-point constants always specify positive values. If negative values are required, the minus sign (–) can be placed in front of the constant to form a constant floating-point expression with a negative value. The minus sign is treated as an arithmetic operator.

The following examples illustrate some of the forms of floating-point constants and expressions:

```
15.75
1.575E1
1575e-2
-0.0025
-2.5e-3
25E-4
```

The integer portion of the floating-point constant can be omitted, as shown in the following examples:

```
.75
.0075e2
-.125
-.175E-2
```

All floating-point constants have type **double**.

2.3.3 Character Constants

A character constant is a letter, digit, punctuation character, or escape sequence enclosed in single quotation marks. The value of a character constant is the numerical representation of the character. Character constants consisting of more than one character or escape sequence are not allowed.

A character constant has the form

'*char*'

where *char* can be any character from the representable character set (including any escape sequence) except a single quotation mark ('), a backslash (\), or a new-line character. To use a single quotation mark or backslash character as a character constant, precede it with a backslash, as shown in Table 2.7. To represent a new-line character, use the escape sequence '\n'.

Table 2.7
Examples of Character Constants

Constant	Value
'a'	Lowercase a
'?'	Question mark
'\b'	Backspace
'\x1B'	ASCII escape character
'\''	Single quotation mark
'\\'	Backslash

Character constants have type **int** and consequently are sign extended in type conversions (see Section 5.7 of Chapter 5, "Expressions and Assignments").

2.3.4 String Literals

A string literal is a sequence of letters, digits, and symbols enclosed in double quotation marks. A string literal is treated as an array of characters; each element of the array is a single character value.

The form of a string literal is

"*characters*"

where *characters* is zero or more characters from the representable character set, excluding the double quotation mark ("), the backslash (\), and the new-line character. To use the new-line character in a string, type a backslash immediately followed by a new-line character. The backslash

causes the new-line character to be ignored, allowing the programmer to form string literals that occupy more than one line. For example, the string literal

```
"Long strings can be bro\
ken into two pieces."
```

is identical to the string

```
"Long strings can be broken into two pieces."
```

To use the double quotation mark or backslash character within a string literal, precede it with a backslash, as shown in the following examples:

```
"This is a string literal."
"Enter a number between 1 and 100 \n or press Return"
"First\\Second"
"\"Yes, I do,\" she said."
"The following line shows a null string:"
""
```

Note that escape sequences (such as \\ and \") can appear in string literals. Each escape sequence counts as a single character.

The characters of a string are stored in order at contiguous memory locations. A null character (**\0**) is automatically appended to mark the end of the string. Each string in a program is considered to be a distinct item; if two identical strings appear in a program, they each receive distinct storage space.

String literals have type **char** [ ]. This means a string is an array whose elements have type **char**. The number of elements in the array is the number of characters in the string literal plus one, since the null character stored after the last character counts as an array element.

2.4 Identifiers

Identifiers are the names you supply for the variables, functions, and labels used in a given program. You create an identifier by declaring it with the associated variable or function. You can then use the identifier in later statements within the program to refer to the given item. (Declarations are described in Chapter 4, "Declarations.")

An identifier is a sequence of one or more letters, digits, or underscores (_) that begins with a letter or underscore. Any number of characters are allowed in a given identifier, but only the first 31 characters are significant to the compiler. (Other programs that read the compiler output, such as the linker, may use fewer characters.) Use leading underscores with care; identifiers beginning with an underscore can conflict with the names of hidden system routines and produce errors.

The following are examples of identifiers:

```
j
cnt
temp1
topofpage
skip12
```

The C compiler considers uppercase and lowercase letters to be separate and distinct characters. Therefore, you can create distinct identifiers that have the same spelling but different cases for one or more of the letters. For example, each of the following identifiers is unique:

```
add
ADD
Add
aDD
```

The C compiler does not allow an identifier that has the same spelling and case as a C language keyword. Keywords are described in Section 2.5.

Note

The linker may further restrict the number and type of characters for globally visible symbols, and, unlike the compiler, the linker may not distinguish between uppercase and lowercase letters. Consult your linker documentation for information on naming restrictions imposed by the linker.

2.5 Keywords

Keywords are predefined identifiers that have special meaning to the C compiler. They can be used only as defined. The names of program items cannot conflict with the keywords listed below:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**while**
do	**if**	**static**	**volatile**

Keywords cannot be redefined. However, you can specify text to be substituted for keywords before compilation by using C preprocessor directives (see Chapter 8, "Preprocessor Directives and Pragmas").

The **const** and **volatile** keywords are reserved for future use but have not yet been implemented.

The following identifiers may be keywords in some implementations. See your system documentation for more information.

cdecl
far
fortran
huge
near
pascal

2.6 Comments

A comment is a sequence of characters that is treated as a single white-space character by the compiler, but is otherwise ignored. A comment has the following form:

/* *characters* */

Here *characters* can be any combination of characters from the representable character set, including new-line characters but excluding the

combination */. This means that comments can occupy more than one line, but they cannot be nested.

Comments can appear anywhere a white-space character is allowed. The compiler ignores the characters of the comment, so keywords can appear in comments without producing errors. Since the compiler treats the comment as a single white-space character, comments cannot appear within a token.

The following examples illustrate some comments:

```
/* Comments can separate and document
   lines of a program. */

/* Comments can contain keywords such as for
   and while. */

/*****************************************
   Comments can occupy several lines.
 *****************************************/
```

Since comments cannot contain nested comments, the following example causes an error:

```
/* You cannot /* nest */ comments */
```

The compiler recognizes the first */, after the word nest, as the end of the comment. The compiler attempts to process the remaining text and produces an error when it cannot do so.

To suppress compilation of a large portion of a program or a program segment that contains comments, use the #**if** preprocessor directive instead of comments (see Section 8.4 of Chapter 8, "Preprocessor Directives and Pragmas").

2.7 Tokens

When the compiler processes a program, it breaks the program down into groups of characters known as "tokens." A token is a unit of program text that has meaning to the compiler and that cannot be broken down further. The operators, constants, identifiers, and keywords described in this chapter are examples of tokens. Punctuation characters such as brackets ([ ]), braces ({ }), angle brackets (< >), parentheses, and commas are also tokens.

Tokens are delimited by white-space characters and by other tokens, such as operators and punctuation symbols. To prevent the compiler from breaking an item down into two or more tokens, white-space characters are prohibited between the characters of identifiers, multicharacter operators, and keywords.

When the compiler interprets tokens, it incorporates as many characters as possible into a single token before moving on to the next token. Because of this behavior, tokens not separated by white space may not be interpreted as expected.

For example, consider the following expression:

```
i+++j
```

In the above example, the compiler first makes the longest possible operator (++) from the three plus signs, and then processes the remaining plus sign as an addition operator (+). This expression is interpreted as `(i++) + (j)`, not `(i) + (++j)`. Use white space and parentheses to clarify your intent in such cases.

Chapter 3
Program Structure

3.1 Introduction

This chapter describes the structure of C language source programs and defines terms used later in this manual to describe the C language. It provides an overview of C language features that are described in detail in other chapters. The syntax and meaning of declarations and definitions are discussed in Chapter 4, "Declarations," and Chapter 7, "Functions." The C preprocessor and pragmas are described in Chapter 8, "Preprocessor Directives and Pragmas."

3.2 Source Program

A C source program is a collection of one or more directives, pragmas, declarations, and/or definitions. "Directives" instruct the C preprocessor to perform specific actions on the text of the program prior to compilation. "Pragmas" are instructions to the compiler that are carried out at compile time.

"Declarations" establish the names and attributes of variables, functions, and types used in the program. "Definitions" are declarations that also define variables and functions. A variable definition gives the initial value of the declared variable, in addition to its name and type. The definition causes storage to be allocated for the variable. A function definition specifies the function body, which is a compound statement containing the declarations and statements that constitute the function. The function definition also gives the function name, formal parameters, and return type.

A source program can have any number of directives, pragmas, declarations, and definitions. Each must have the appropriate syntax as described in this manual, and each can appear in any order in the program (subject to the rules outlined throughout this manual), although the order affects how variables and functions can be used in the program (see Section 3.5, "Lifetime and Visibility").

A nontrivial program always contains at least one definition, a function definition. The function defines the action to be taken by the program. The following example illustrates a simple C source program:

Example

```
int x = 1;                          /* Variable definitions */
int y = 2;

extern int printf(char *,...);/* Function declaration */

main ()                             /* Function definition
                                       for main function */
{
        int z;                      /* Variable declarations */
        int w;

        z = y + x;                  /* Executable statements */
        w = y - x;
        printf("z= %d \nw= %d \n", z, w);
}
```

This source program defines the function named `main` and declares the function `printf`. The variables `x` and `y` are defined with variable definitions; the variables `z` and `w` are just declared.

3.3 Source Files

Source programs can be divided into one or more source files. A C source file is a text file that contains all or part of a C source program; it may, for example, contain just a few of the functions needed by the program. When the source program is compiled, the individual source files that make up the program must be compiled individually and then linked. Separate source files can also be combined to form larger source files before compilation by using the **#include** directive, discussed in Chapter 8, "Preprocessor Directives and Pragmas."

A source file can contain any combination of complete directives, pragmas, declarations, and definitions. Items such as function definitions or large data structures cannot be split between source files.

A source file need not contain any executable statements. It is sometimes useful to place variable definitions in one source file and then declare references to these variables in other source files that use them. This makes the definitions easy to find and modify, if necessary. For the same reason, manifest constants and macros (discussed in Chapter 8, "Preprocessor Directives and Pragmas") are often organized into separate "include" files and inserted into source files where required.

Directives in a source file apply to that source file and its include files only. Moreover, each directive applies only to the portion of the file following the directive. If a common set of directives is to be applied to a source program, then all source files in the program must contain these directives.

Pragmas usually take effect over a specific region of a source file. However, the specific compiler action that is defined by a pragma is determined by the implementation. For a discussion of the effects of particular pragmas, see your system documentation.

The following is an example of a C source program contained in two source files. The `main` and `max` functions are assumed to be in separate files, and execution of the program is assumed to begin with the `main` function.

Example

```
/*************************************************************
        Source file 1 - main function
 *************************************************************/

#define ONE     1
#define TWO     2
#define THREE   3

extern int max(int, int);       /* Function declaration */

main ()                         /* Function definition */
{
        int w = ONE, x = TWO, y = THREE;
        int z = 0;
        z = max(x,y);
        w = max(z,w);
}

/*************************************************************
        Source file 2 - max function
 *************************************************************/

int max (a, b)                  /* Function definition */
int a, b;
{
        if ( a > b )
                return (a);
        else
                return (b);
}
```

In the first source file, the function `max` is declared without being defined. This is known as a "forward declaration." The function definition for `main` includes function calls to `max`.

The lines beginning with a number sign (#) are preprocessor directives. These directives instruct the preprocessor to replace the identifiers ONE, TWO, and THREE with the specified number in the first source file. The directives do not apply to the second source file.

The second source file contains the function definition for `max`. This definition satisfies the calls to `max` in the first source file. Once the source files are compiled, they can be linked and executed as a single program.

3.4 Program Execution

Every program has a primary (main) program function. In C, the primary program function must be named **main**. The **main** function serves as the starting point for program execution and usually controls execution of the program by directing the calls to other functions in the program. A program usually stops executing at the end of the **main** function, although it can stop at other points in the program, depending on the execution environment.

The source program usually has more than one function, each designed to perform one or more specific tasks. The **main** function can call these functions to perform the tasks. When a function is called, execution begins at the first statement in the called function. The function returns control when a **return** statement is executed or the end of the function is encountered.

All functions, including the **main** function, can be declared to have parameters. Functions called by other functions receive values for the parameters from the calling functions. Parameters of the **main** function can be declared to receive values passed to the **main** function from outside the program. For example, they can receive values from the command line when the program is executed.

When the **main** function takes parameters, C requires the first two parameters to be named **argc** and **argv**. The **argc** parameter is declared to hold the total number of arguments passed to the **main** function. The **argv** parameter is declared as an array of pointers, each element of which points to a string representation of an argument passed to the **main** function.

Traditionally, the third parameter to the **main** function (if there is a third parameter) is given the name **envp**. The C language does not require this name, however. The **envp** parameter is a pointer to a table of string values that set up the environment in which the program executes.

The operating system supplies values for the **argc**, **argv**, and **envp** parameters, and the user supplies the actual arguments to the **main** function. The argument-passing convention in use on a particular system is determined by the operating system rather than by the C language. For more information, see your system documentation.

Formal parameters to functions must be declared when the function is defined. Function definitions are described in more detail in Section 7.2 of Chapter 7, "Functions." Function declarations are discussed in Section 4.5 of Chapter 4, "Declarations."

3.5 Lifetime and Visibility

Two concepts, "lifetime" and "visibility," are important in understanding the structure of a C program. The lifetime of a variable or function can be either "global" or "local." An item with a global lifetime has storage and a defined value throughout the duration of the program; an item with a local lifetime is allocated new storage each time the "block" in which it is defined or declared is entered. When the block is exited, the local item loses its storage, and hence its value. Blocks are defined and discussed below.

An item is said to be "visible" in a block or source file if the type and name of the item are known in the block or source file. An item can also be "globally visible," which means that it is visible, or can be made visible through appropriate declarations, throughout all the source files that constitute the program. Visibility between source files (also known as "linkage") is discussed in greater detail in Section 4.6 of Chapter 4, "Declarations."

A block is a compound statement. Compound statements consist of declarations and statements, as described in Section 6.3 of Chapter 6, "Statements." The bodies of C functions are compound statements. Blocks can be nested; function bodies frequently contain blocks, which in turn can contain other blocks.

Declarations and definitions within blocks occur at the "internal level." Declarations and definitions outside all blocks occur at the "external level."

Both variables and functions can be *declared* at the external level or at the internal level. Variables can also be *defined* at the internal level, but functions can only be defined at the external level.

All functions have global lifetimes, regardless of where they are declared. Variables declared at the external level always have global lifetimes. Variables declared at the internal level usually have local lifetimes; however, the storage-class specifiers **static** and **extern** can be applied to declare global variables or references to global variables within a block. See Section 4.6 of Chapter 4, "Declarations," for a discussion of these options.

Variables declared or defined at the external level are visible from the point at which they are declared or defined to the end of the source file. These variables can be made visible in other source files with appropriate declarations, as described in Section 4.6, "Storage Classes." However, variables that are given **static** storage class at the external level are visible only within the source file in which they are defined.

In general, variables declared or defined at the internal level are visible from the point at which they are first declared or defined to the end of the block in which the definition or declaration appears. These variables are called local variables. If a variable declared inside a block has the same name as a variable declared at the external level, the block definition supersedes the external level definition of the variable for the duration of the block. The visibility of the external level variable is restored when the block is exited.

Block visibility can nest. This means that a block nested inside another block can contain declarations that redefine variables declared in the outer block. The redefinition of the variable holds in the inner block, but the original definition is restored when control returns to the outer block. Variables from outer blocks are visible inside all inner blocks, as long as they are not redefined in the inner blocks.

Functions with **static** storage class are visible only in the source file in which they are defined. All other functions are globally visible. For more information on function declarations, see Section 4.5 of Chapter 4, "Declarations."

Table 3.1 summarizes the main factors that determine the lifetime and visibility of functions and variables. The table is not, however, intended to cover all cases. Refer to the above discussion and to Section 4.6, "Storage Classes," for more information.

Table 3.1

Summary of Lifetime and Visibility

Level	Item	Storage Class Specifier	Lifetime	Visibility
External	Variable declaration	**static**	Global	Restricted to single source file
	Variable declaration	**extern**	Global	Remainder of source file
	Function declaration or definition	**static**	Global	Restricted to single source file
	Function declaration or definition	**extern**	Global	Remainder of source file
Internal	Variable definition or declaration	**extern** or **static**	Global	Block
	Variable definition or declaration	**auto** or **register**	Local	Block

The following program example illustrates blocks, nesting, and visibility of variables:

Example

```
/* i defined at external level */
int i = 1;

/* main function defined at external level */
main ()
{

    /* prints 1 (value of external level i) */
    printf("%d\n", i);

    /* first nested block */
    {

        /* i and j defined at internal level */
```

```
        int i = 2, j = 3;

        /* prints 2, 3 */
        printf("%d\n%d\n", i, j);

        /* second nested block */
        {

            /* i is redefined */
            int i = 0;

            /* prints 0, 3 */
            printf("%d\n%d\n", i, j);

        /* end of second nested block */
        }

        /* prints 2 (outer definition restored) */
        printf("%d\n", i);

    /* end of first nested block */
    }

    /* prints 1 (external level definition restored) */
    printf("%d\n", i);
}
```

In this example, there are four levels of visibility: the external level and three block levels. Assuming that the function `printf` is defined elsewhere in the program, the `main` function prints out the values 1, 2, 3, 0, 3, 2, 1.

3.6 Naming Classes

In any C program, identifiers are used to refer to many different kinds of items. When you write a C program, you provide identifiers for the functions, variables, formal parameters, union members, and other items the program uses. C allows you to use the same identifier for more than one program item, as long as you follow the rules outlined in this section.

The compiler sets up "naming classes" to distinguish between the identifiers for different kinds of items. The names within each class must be unique to avoid conflict, but an identical name can appear in one or more naming classes. This means that you can use the same identifier for

two or more different items if the items are in different naming classes. The context of a given identifier in the program allows the compiler to resolve the reference without ambiguity.

The kinds of items you can name in C programs, and the rules for naming them, are described as follows:

Items	Naming Class
Variables and functions	The names of variables and functions are in a naming class with formal parameters and enumeration constants. Variable and function names must, therefore, be distinct from other names in this class with the same visibility. However, variable names can be redefined within program blocks, as described in Section 3.5, "Lifetime and Visibility." Function names can also be redefined in this manner.
Formal parameters	The names of formal parameters to a function are grouped with the names of the function's variables, so the formal parameter names should be distinct from the variable names. Redeclaring formal parameters within the function causes an error.
Enumeration constants	Enumeration constants are in the same naming class as variable and function names. This means that names of enumeration constants must be distinct from all variable and function names with the same visibility, and distinct from the names of other enumeration constants with the same visibility. However, like variable names, the names of enumeration constants have nested visibility, meaning that they can be redefined within blocks. See Section 3.5, "Lifetime and Visibility."
typedef names	The names of types defined with **typedef** are in a naming class with variable and function names. They must, therefore, be distinct from all variable and function

	names with the same visibility, and also from the names of formal parameters and enumeration constants. Like variable names, names used for **typedef** types can be redefined within program blocks. See Section 3.5, "Lifetime and Visibility."
Tags	Enumeration, structure, and union tags are grouped together in a single naming class. Each enumeration, structure, or union tag must be distinct from other tags with the same visibility. Tags do not conflict with any other names.
Members	The members of each structure and union form a naming class. The name of a member must, therefore, be unique within the structure or union, but it does not have to be distinct from any other name in the program, including names of members of different structures and unions.
Statement labels	Statement labels form a separate naming class. Each statement label must be distinct from all other statement labels in the same function. Statement labels do not have to be distinct from any other names or from label names in other functions.

Example

```
struct student {
        char student[20];
        int class;
        int id;
        } student;
```

Structure tags, structure members, and variable names are in three different naming classes, so no conflict occurs among the three items named `student` in the above example. The compiler determines how to interpret each occurrence of `student` by its context in the program. For example, when `student` appears after the **struct** keyword, it is known to be a structure tag. When `student` appears after a member-selection operator (`->` or `.`), the name refers to the structure member. In other contexts, the identifier `student` refers to the structure variable.

Chapter 4
Declarations

4.1 Introduction

This chapter describes the form and constituents of C declarations for variables, functions, and types. C declarations have the form

[[*sc-specifier*]] [[*type-specifier*]] *declarator*[[=*initializer*]] [[,*declarator*[[=*initializer*]]...]]

where *sc-specifier* is a storage-class specifier, *type-specifier* is the name of a defined type, *declarator* is an identifier that can be modified to declare a pointer, array, or function, and *initializer* gives a value or sequence of values to be assigned to the variable being declared.

All C variables must be explicitly declared before they are used. C functions can be declared explicitly in a function declaration or implicitly by calling the function before it is declared or defined.

The C language defines a standard set of data types. You can add to that set by declaring new data types based on types already defined. You can declare arrays, data structures, and pointers to both variables and functions.

C declarations require one or more *declarators.* A declarator is an identifier that can be modified with brackets ([ ]), asterisks (*), or parentheses to declare an array, pointer, or function type. When you declare simple variables (such as character, integer, and floating-point values), or structures and unions of simple variables, the declarator is just an identifier.

Four storage-class specifiers are defined in C: **auto**, **extern**, **register**, and **static**. The storage-class specifier of a declaration affects how the declared item is stored and initialized and which portions of a program can reference it. The location of the declaration within the source program and the presence or absence of other declarations of the variable are also important factors in determining the visibility of variables.

Function declarations are presented in Section 4.5. For information on function definitions, see Section 7.2.

4.2 Type Specifiers

The C language provides definitions for a set of basic data types, called "fundamental" types. Their names are listed in Table 4.1.

Table 4.1
Fundamental Types

Integral Types[a]	Floating-Point Types[a]	Other
signed char	**float**	**void**[b]
signed int	**double** (also called **long float**)	
signed short int		
signed long int		
unsigned char		
unsigned int		
unsigned short int		
unsigned long int		

[a] Used to declare variables and function return types

[b] Used only to declare function return types

Enumeration types are also considered fundamental types. Type specifiers for enumeration types are discussed in Section 4.8.1. The **signed char**, **signed int**, **signed short int**, and **signed long int** types, together with their **unsigned** counterparts, are called "integral" types. The **float** and **double** type specifiers refer to "floating-point" types. Variable and function declarations can use any of the integral or floating-point type specifiers.

The **void** type can be used only to declare functions that return no value. Function types are discussed in Section 4.5, "Function Declarations."

You can create additional type specifiers with **typedef** declarations, discussed in Section 4.8.2.

Type specifiers are commonly abbreviated, as shown in Table 4.2. Integral types are signed by default. Thus, if the **unsigned** keyword is omitted from the type specifier, the integral type is signed, even if the **signed** keyword is not specified.

In some implementations, a compiler option may be available to change the default for the **char** type from signed to unsigned type. When such an option is in effect, the abbreviation **char** has the same meaning as **unsigned char**, and the **signed** keyword must be used to declare a signed character value.

Table 4.2

Type Specifiers and Abbreviations

Type Specifier	Abbreviations
signed char[a]	**char**
signed int	**signed, int**
signed short int	**short, signed short**
signed long int	**long, signed long**
unsigned char[b]	--
unsigned int	**unsigned**
unsigned short int	**unsigned short**
unsigned long int	**unsigned long**
float	--
long float	**double**

[a] When the **char** type is made unsigned by default (through the use of a compiler option), **signed char** cannot be abbreviated.

[b] When the **char** type is made unsigned by default (through the use of a compiler option), **unsigned char** can be abbreviated as **char**.

Note

This manual generally uses the abbreviated forms listed in Table 4.2 rather than the long forms of the type specifiers and assumes that the **char** type is signed by default. Therefore, throughout this manual, **char** stands for **signed char**.

Table 4.3 summarizes the storage associated with each fundamental type and gives the range of values that can be stored in a variable of each type. Since the **void** type does not apply to variables, it is not included in the table.

Table 4.3

Storage and Range of Values for Fundamental Types

Type	Storage	Range of Values (Internal)
char	1 byte	−128 to 127
int	implementation dependent	
short	2 bytes	−32768 to 32767
long	4 bytes	−2,147,483,648 to 2,147,483,647
unsigned char	1 byte	0 to 255
unsigned	implementation dependent	
unsigned short	2 bytes	0 to 65535
unsigned long	4 bytes	0 to 4,294,967,295
float	4 bytes	IEEE standard notation; discussed below
double	8 bytes	IEEE standard notation; discussed below

The **char** type is used to store a letter, digit, or symbol from the representable character set. The integer value of a character is the ASCII code corresponding to that character. Since the **char** type is interpreted as a signed 1-byte integer, values in the range −128 to 127 are permitted for

char variables, although only the values from 0 to 127 have character equivalents. Similarly, the **unsigned char** type can store values in the range 0 to 255.

Note that the storage and range associated with the **int** and **unsigned int** types are not defined by the C language. Instead, the size of an **int** (signed or unsigned) corresponds to the natural size of an integer on a given machine. For example, on a 16-bit machine the **int** type is usually 16 bits, or 2 bytes. On a 32-bit machine the **int** type is usually 32 bits, or 4 bytes. Thus, the **int** type is equivalent either to the **short int** or the **long int** type, depending on the implementation. Similarly, the **unsigned int** type is equivalent either to the **unsigned short** or **unsigned long** type.

The **int** and **unsigned int** type specifiers are widely used in C programs because they allow a particular machine to handle integer values in the most efficient way for that machine. However, since the size of the **int** and **unsigned int** types varies, programs that depend on a specific **int** size may be nonportable. Expressions involving the **sizeof** operator (discussed in Section 5.3.4) can be used in place of hard-coded data sizes to increase the portability of the code.

The type specifiers **int** and **unsigned int** (or simply **unsigned**) are used to define certain features of the C language (for instance, for defining the **enum** type later in Section 4.8.1). In these cases, the definition of **int** and **unsigned int** for a particular implementation determines the actual storage.

Range of Values

The range of values for a variable lists the minimum and maximum values that can be represented *internally* in a given number of bits. However, because of C's conversion rules (discussed in detail in Chapter 5, "Expressions and Assignments"), it is not always possible to use the maximum or minimum for a constant of a given type in an expression.

For example, the constant expression –32768 consists of the arithmetic negation operator (–) applied to the constant value 32768. Since 32768 is too large to represent as a **short**, it is given **long** type, and the constant expression –32768 consequently has **long** type. The value –32768 can only be represented as a **short** by type casting it to the **short** type. No information is lost in the type cast, since –32768 can be represented internally in 2 bytes of storage space.

Similarly, a value such as 65000 can only be represented as an **unsigned short** by type casting the value to **unsigned short** type or by giving the value in octal or hexadecimal notation. The value 65000 in decimal notation is considered a signed constant, and is given **long** type because 65000 does not fit into a **short.** This **long** value can then be cast to the **unsigned short** type without loss of information, since 65000 will fit into 2 bytes of storage space when it is stored as an unsigned number.

Octal and hexadecimal constants may have either signed or unsigned type, depending on their size (see Section 2.3.1 for more information). However, the method used for assigning types to these constants ensures that they always behave like unsigned integers in type conversions.

Floating-point numbers use the IEEE (Institute of Electrical and Electronics Engineers, Inc.) format. Values with **float** type have 4 bytes, consisting of a sign bit, a 7-bit excess 127 binary exponent, and a 24-bit mantissa. The mantissa represents a number between 1.0 and 2.0. Since the high-order bit of the mantissa is always 1, it is not stored in the number. This representation gives a range of approximately 3.4E–38 to 3.4E+38.

Values with **double** type have 8 bytes. The format is similar to the **float** format, except that the exponent is 11 bits excess 1023, and the mantissa has 52 bits, plus the implied high-order 1 bit. This gives a range of approximately 1.7E–308 to 1.7E+308.

4.3 Declarators

Syntax

identifier
declarator[ ]
declarator[*constant-expression*]
**declarator*
declarator( )
declarator(*arg-type-list*)
(*declarator*)

C allows the programmer to declare *arrays* of values, *pointers* to values, and *functions returning* values of specified types. To declare these items, you must use a *declarator.*

A declarator is an identifier, possibly modified with brackets ([ ]), parentheses, and asterisks (*), to declare an array, pointer, or function type. Declarators appear in the pointer, array, and function declarations described in later sections of this chapter (Sections 4.4.6, 4.4.5, and 4.5, respectively). This section discusses the rules for forming and interpreting declarators.

4.3.1 Pointer, Array, and Function Declarators

When a declarator consists of an unmodified identifier, the item being declared has an unmodified type. Asterisks (*) can appear to the left of an identifier, modifying it to a *pointer* type. If the identifier is followed by brackets ([ ]), the type is modified to an *array* type. If the identifier is followed by parentheses, the type is modified to a *function returning* type.

A declarator does not constitute a complete declaration; a type specifier must be included as well. The type specifier gives the type of the elements for an array type, the type of object addressed by a pointer type, and the return type of a function.

The sections on pointer, array, and function declarations later in this chapter discuss each type of declaration in detail (see Sections 4.4.6, 4.4.5, and 4.5, respectively). The following examples illustrate the simplest forms of declarators:

Examples

```
int list[20];           /* Example 1 */

char *cp;               /* Example 2 */

double func(void);      /* Example 3 */
```

The above examples declare the following:

1. An array of **int** values (`list`)
2. A pointer to a **char** value (`cp`)
3. A function with no arguments returning a **double** value (`func`)

4.3.2 Complex Declarators

Any declarator can be enclosed in parentheses. Parentheses are typically used to specify a particular interpretation of a "complex" declarator, as discussed below. A "complex" declarator is an identifier qualified by more than one array, pointer, or function modifier.

Various combinations of the array, pointer, and function modifiers can be applied to a single identifier. Some combinations are illegal. An array cannot be composed of functions, and a function cannot return an array or a function.

In interpreting complex declarators, brackets and parentheses (to the right of the identifier) take precedence over asterisks (to the left of the identifier). Brackets and parentheses have the same precedence and associate left to right. The type specifier is applied as the last step, when the declarator has been fully interpreted. Parentheses can be used to override the default association order in a way that forces a particular interpretation.

A simple rule that can be helpful in interpreting complex declarators is to read them "from the inside out." Start with the identifier and look to the right for brackets or parentheses. Interpret these (if any), then look to the left for asterisks. If you encounter a right parenthesis at any stage, go back and apply these rules to everything within the parentheses before proceeding. As the last step, apply the type specifier. To illustrate this rule, the steps are numbered in order in the following example:

```
char *(*(*var)())[10];
 ^    ^ ^ ^ ^  ^  ^
 7    6 4 2 1  3  5
```

1. The identifier `var` is declared as
2. a pointer to
3. a function returning
4. a pointer to
5. an array of 10 elements, which are
6. pointers to
7. **char** values.

The following examples provide further illustration and show how parentheses can affect the meaning of a declaration:

Examples

```
/******************** Example 1 ********************/

        /* array of pointers to int values */
int *var[5];

/******************** Example 2 ********************/

       /* pointer to array of int values */
int (*var)[5];

/******************** Example 3 ********************/

      /* function returning pointer to long */
long *var(long,long);

/******************** Example 4 ********************/

      /* pointer to function returning long */
long (*var)(long,long);

/******************** Example 5 ********************/

      /* array of pointers to functions
         returning structures */
struct both {
        int a;
        char b;
        } ( *var[5] )( struct both, struct both );

/******************** Example 6 ********************/

      /* function returning pointer
         to an array of 3 double values */
double ( *var( double (*)[3] ) )[3];

/******************** Example 7 ********************/

      /* array of arrays of pointers
         to pointers to unions */
union sign {
        int x;
        unsigned y;
        } **var[5][5];

/******************** Example 8 ********************/

      /* array of pointers to arrays
         of pointers to unions */
union sign *(*var[5])[5];
```

Example	Description
1	In the first example, the array modifier has higher priority than the pointer modifier, so `var` is declared to be an array. The pointer modifier applies to the type of the array elements; the elements are pointers to **int** values.
2	In the second example, parentheses alter the meaning of the declaration in the first example. Now the pointer modifier has higher priority than the array modifier, and `var` is declared to be a pointer to an array of five **int** values.
3	Function modifiers also have higher priority than pointer modifiers, so the third example declares `var` to be a function returning a pointer to a **long** value. The function is declared to take two **long** values as arguments.
4	The fourth example is similar to the second example. Parentheses give the pointer modifier higher priority than the function modifier, and `var` is declared to be a pointer to a function returning a **long** value. Again, the function takes two **long** arguments.

5	The elements of an array cannot be functions, but the fifth example demonstrates how to declare an array of pointers to functions instead. In this example, `var` is declared to be an array of five pointers to functions returning structures with two members. The arguments to the functions are declared to be two structures with the same structure type, `both`. Note that the parentheses surrounding `*var[5]` are required. Without them, the declaration is an illegal attempt to declare an array of functions, as shown below:
	`/* ILLEGAL */`
	`struct both *var[5]( struct both, struct both );`

6	The sixth example shows how to declare a function returning a pointer to an array, since functions returning arrays are illegal. Here `var` is declared to be a function returning a pointer to an array of three **double** values. The function `var` takes one argument; the argument, like the return value, is a pointer to an array of three **double** values. The argument type is given by a complex abstract declarator. The parentheses around the asterisk in the argument type are required; without them, the argument type would be an array of three

pointers to **double** values. For a discussion and examples of abstract declarators, see Section 4.9, "Type Names."

7 A pointer can point to another pointer, and an array can contain array elements, as the seventh example shows. Here `var` is an array of five elements. Each element is a five-element array of pointers to pointers to unions with two members.

8 The eighth example shows how the placement of parentheses alters the meaning of the declaration. In this example, `var` is a five-element array of pointers to five-element arrays of pointers to unions.

4.3.3 Declarators with Special Keywords

Your implementation of Microsoft C may include the following special keywords:

cdecl
far
fortran
huge
near
pascal

These keywords are used to modify the meaning of variable and function declarations. See your system documentation for a full discussion of the effects of these special keywords.

When a special keyword occurs in a declarator, it modifies the item immediately to the right of the keyword. More than one special keyword can be applied to the same item. For example, a function identifier might be modified with both the **far** and **pascal** keywords. The order of the keywords in this case does not matter (that is, **far pascal** and **pascal far** have the same effect).

Two or more special keywords can be used in different parts of the declaration to modify the meaning of the declaration. For example, the following declaration contains two occurrences of the **far** keyword:

```
int far * pascal far func(void);
```

The function identifier `func` is modified with the **pascal** and **far** keywords. The return value of `func` is declared to be a **far** pointer to an **int** value.

As in any C declaration, parentheses can be used to override the default interpretation of the declaration. The rules governing complex declarators (discussed in the previous section) apply to declarations using the special keywords as well.

The following examples show the use of special keywords in declarations:

Examples

```
/******************** Example 1 ********************/

int huge database[65000];

/******************** Example 2 ********************/

char * far * x;

/******************** Example 3 ********************/

double near cdecl calc(double,double);
double cdecl near calc(double,double);

/******************** Example 4 ********************/

char far fortran initlist[INITSIZE];
char far *nextchar, far *prevchar, far *currentchar;

/******************** Example 5 ********************/

char far *(far *getint)(int far *);
 ^    ^       ^       ^     ^    ^
 6    5       2       1     3    4
```

Example 1 declares a **huge** array named `database` with 65000 **int** elements. The **huge** keyword modifies the array declarator.

In Example 2, the **far** keyword modifies the asterisk to its right, making `x` a **far** pointer to a pointer to **char**. The declaration could be expressed equivalently in the following manner:

```
char * (far *x);
```

Example 3 shows two equivalent declarations. Both declare `calc` as a function with the **near** and **cdecl** attributes.

Example 4 also shows two declarations: the first declares a **far fortran** array of characters named `initlist`, and the second declares three **far** pointers named `nextchar`, `prevchar`, and `currentchar`. The pointers might be used to store the addresses of characters in the `initlist` array. Note that the **far** keyword must be repeated before each declarator.

Example 5 shows a more complex declaration with several occurrences of the **far** keyword. The steps in interpreting this declaration are as follows:

1. The identifier `getint` is declared as a
2. **far** pointer to
3. a function taking
4. a single argument that is a **far** pointer to an **int** value
5. and returning a **far** pointer to a
6. **char** value.

Note that the **far** keyword always modifies the item immediately to its right.

4.4 Variable Declarations

This section describes the form and meaning of variable declarations. In particular, it explains how to declare the following:

Type of Variable	Description
Simple variables	Single-value variables with integral or floating-point type.
Enumeration variables	Simple variables with integral type that hold one value from a set of named integer constants.
Structures	Variables composed of a collection of values that may have different types.
Unions	Variables composed of several values of different types occupying the same storage space.

Arrays	Variables composed of a collection of elements with the same type.
Pointers	Variables that point to other variables. These variables contain variable locations (in the form of addresses) instead of values.

The variable declarations discussed in this section have the general form

[[*sc-specifier*]] *type-specifier declarator* [[, *declarator...*]]

where *type-specifier* gives the data type of the variable and *declarator* is the variable's name, possibly modified to declare an array or a pointer type. More than one variable can be defined in the declaration by giving multiple declarators, separated by commas.

The *sc-specifier* gives the storage class of the variable. In some contexts, variables can be initialized when they are declared. For information on storage classes and initialization, see Sections 4.6 and 4.7, respectively.

4.4.1 Simple Variable Declarations

Syntax

type-specifier identifier [[, *identifier...*]];

A declaration for a simple variable defines the variable's name and type; it can also define the variable's storage class, as described in Section 4.6. The variable's name is the *identifier* given in the declaration. The *type-specifier* gives the name of a defined data type, as described below.

You can define several variables in the same declaration by giving a list of identifiers separated by commas (,). Each identifier in the list names a variable. All variables defined in the declaration have the same type.

Examples

```
int x;                          /* Example 1 */

unsigned long reply, flag;      /* Example 2 */

double order;                   /* Example 3 */
```

The first example defines a simple variable `x`. This variable can hold any value in the set defined by the **int** type in a particular implementation.

The second example defines two variables, `reply` and `flag`. Both variables have **unsigned long** type and hold unsigned integer values.

The third example defines a variable `order` that has **double** type. Floating-point values can be assigned to this variable.

4.4.2 Enumeration Declarations

Syntax

enum [[*tag*]] { *enum-list*} *identifier* [[, *identifier*...]];
enum *tag identifier* [[, *identifier*...]];

An enumeration declaration gives the name of the enumeration variable and defines a set of named integer constants (the "enumeration set"). A variable declared to have enumeration type stores any one of the values of the enumeration set defined by that type. The integer constants of the enumeration set have **int** type; thus, the storage associated with an enumeration variable is the storage required for a single **int** value.

Enumeration declarations begin with the **enum** keyword and have two forms, as shown above. In the first form, the values and names of the enumeration set are specified in the *enum-list*, described in detail below. The optional *tag* is an identifier that names the enumeration type defined by the *enum-list*. The *identifier* names the enumeration variable. More than one enumeration variable can be defined in the declaration.

The second form uses an enumeration *tag* to refer to an enumeration type. The *enum-list* does not appear in this type of declaration because the enumeration type is defined elsewhere. An error is generated if the given *tag* does not refer to a defined enumeration type or if the named type is not currently visible.

An *enum-list* has the following form:

identifier [[= *constant-expression*]]
[[, *identifier* [[= *constant-expression*]]]]
.
.
.

Each *identifier* names a value of the enumeration set. By default, the first identifier is associated with the value 0, the next identifier is associated with the value 1, and so on through the last identifier appearing in the declaration. The name of an enumeration constant is equivalent to its value.

The phrase "= *constant-expression*" overrides the default sequence of values. An identifier followed by the phrase "= *constant-expression*" is associated with the value given by *constant-expression*. The *constant-expression* must have **int** type and can be negative. The next identifier in the list is associated with the value of "*constant-expression* + 1", unless it is explicitly given another value.

An enumeration set can contain duplicate constant values, but each identifier in an enumeration list must be unique; that is, it must be different from all other enumeration identifiers with the same visibility. For example, the value 0 could be given to two different identifiers, `null` and `zero`, in the same set. The identifiers in the list must also be distinct from other identifiers with the same visibility, including ordinary variable names and identifiers in other enumeration lists. Enumeration tags must be distinct from other enumeration, structure, and union tags with the same visibility.

Examples

```
/******************** Example 1 ********************/

enum day {
        saturday,
        sunday = 0,
        monday,
        tuesday,
        wednesday,
        thursday,
        friday
        } workday;

/******************** Example 2 ********************/

enum day today = wednesday;
```

The first example defines an enumeration type named `day` and declares a variable named `workday` with that enumeration type. The value 0 is associated with `saturday` by default. The identifier `sunday` is explicitly set to 0. The remaining identifiers are given the values 1 through 5 by default.

In the second example, a value from the set is assigned to the variable `today`. Note that the name of the enumeration constant is used to assign the value. Since the `day` enumeration type was previously declared, only the enumeration tag is necessary in this declaration.

4.4.3 Structure Declarations

Syntax

struct [[*tag*]] { *member-declaration-list*} *declarator* [[, *declarator*...]];
struct *tag declarator* [[, *declarator*...]];

A structure declaration defines the name of the structure variable and specifies a sequence of variable values (called "members" of the structure) that can have different types. A variable with structure type holds the entire sequence defined by that type.

Structure declarations begin with the **struct** keyword and have two forms, as shown above. In the first form, the types and names of the structure members are specified in the *member-declaration-list*, described in detail below. The optional *tag* is an identifier that names the structure type defined by the *member-declaration-list*.

Each *declarator* gives the name of a structure variable. The *declarator* may also modify the type of the variable to a pointer to the structure type, an array of structures, or a function returning a structure.

The second form uses a structure *tag* to refer to a structure type. The *member-declaration-list* does not appear in this type of declaration because the structure type is defined elsewhere. The structure type definition must be visible for a *tag* declaration to be used, and the definition must appear prior to the *tag* declaration, unless the *tag* is used to declare a pointer variable or a **typedef** structure type. These declarations can use a structure *tag* before the structure type is defined, as long as the structure definition is visible to the declaration.

A *member-declaration-list* is a list of one or more variable or bit-field declarations. Each variable declared in the *member-declaration-list* is defined as a member of the structure type. Variable declarations within member declaration lists have the same form as the variable declarations discussed in this chapter, except that the declarations do not contain storage-class specifiers or initializers. The structure members can have any variable type: fundamental, array, pointer, union, or structure.

A member cannot be declared to have the type of the structure in which it appears. However, a member can be declared as a pointer to the structure type in which it appears, allowing you to create linked lists of structures.

Bit Fields

A bit-field declaration has the following form:

type-specifier [[*identifier*]] : *constant-expression*;

The bit field consists of the number of bits specified by *constant-expression*. The *type-specifier* for a bit-field declaration must specify an unsigned integral type, and the *constant-expression* must be a non-negative integer value. Arrays of bit fields, pointers to bit fields, and functions returning bit fields are not allowed. The optional *identifier* names the bit field. An unnamed bit field whose width is specified as 0 has a special function: it guarantees that storage for the member following it in the declaration list begins on an **int** boundary.

The identifiers in the structure declaration list must be unique within that list. It is not necessary for the identifiers in the list to be distinct from ordinary variable names or from identifiers in other structure declaration lists. Structure tags must be distinct from other structure, union, and enumeration tags having the same visibility.

Storage

Structure members are stored sequentially in the same order in which they are declared: the first member has the lowest memory address and the last member the highest. The storage for each member begins on a memory boundary appropriate to its type. Therefore, unnamed blanks can occur between the members of a structure in memory.

Bit fields are not stored across boundaries of their declared type. For example, a bit field declared with **unsigned int** type is either packed into the space remaining in the previous **unsigned int** or it begins a new **unsigned int**.

Examples

```
/******************** Example 1 ********************/

struct {
            float x,y;
    } complex;

/******************** Example 2 ********************/

struct employee {
            char name[20];
            int id;
            long class;
    } temp;

/******************** Example 3 ********************/

struct employee student, faculty, staff;

/******************** Example 4 ********************/

struct sample {
            char c;
            float *pf;
            struct sample *next;
    } x;

/******************** Example 5 ********************/

struct {
            unsigned icon : 8;
            unsigned color : 4;
            unsigned underline : 1;
            unsigned blink : 1;
    } screen[25][80];
```

The first example defines a structure variable named `complex`. This structure has two members with **float** type, `x` and `y`. The structure type is not named.

The second example defines a structure variable named `temp`. The structure has three members: `name`, `id`, and `class`. The `name` member is a 20-element array and `id` and `class` are simple members with **int** and **long** type, respectively. The identifier `employee` is the structure tag.

The third example defines three structure variables: `student`, `faculty`, and `staff`. Each structure has the same list of three members. The members are declared to have the structure type `employee`, defined in the previous example.

The fourth example defines a structure variable named `x`. The first two members of the structure are a **char** variable and a pointer to a **float** value. The third member, `next`, is declared as a pointer to the structure type being defined (`sample`).

The fifth example defines a two-dimensional array of structures named `screen`. The array contains 2000 elements, and each element is an individual structure containing four bit-field members: `icon`, `color`, `underline`, and `blink`.

4.4.4 Union Declarations

Syntax

union [[*tag*]] { *member-declaration-list*} *declarator* [[, *declarator*...]];
union *tag declarator*[[, *declarator*...]];

A union declaration defines the name of the union variable and specifies a set of variable values, called "members" of the union, that can have different types. A variable with union type stores any single value defined by that type.

Union declarations have the same forms as structure declarations except that they begin with the **union** keyword instead of the **struct** keyword. The same rules govern structure and union declarations, except that bit-field members are not allowed in unions.

The storage associated with a union variable is the storage required for the longest member of the union. When a smaller member is used, the union variable may contain unused memory space. All members are stored in the same memory space and start at the same address. The stored value is overwritten each time a value is assigned to a different member.

Examples

```
/******************** Example 1 ********************/

union sign {
        int svar;
        unsigned uvar;
    } number;

/******************** Example 2 ********************/

union {
        char *a, b;
        float f[20];
    } jack;

/******************** Example 3 ********************/

union {
        struct {
                char icon;
                unsigned color : 4;
        } window1, window2, window3, window4;
    } screen[25][80];
```

The first example defines a union variable named `number` that has two members: `svar`, a signed integer, and `uvar`, an unsigned integer. This declaration allows the current value of `number` to be stored as either a signed or an unsigned value. The union type is named `sign`.

The second example defines a union variable named `jack`. The members of the union are, in order of their declaration, a pointer to a **char** value, a **char** value, and an array of **float** values. The storage allocated for `jack` is the storage required for the 20-element array `f`, since `f` is the longest member of the union. The union type is unnamed.

The third example defines a two-dimensional array of unions named `screen`. The array contains 2000 elements. Each element is an individual union with four members: `window1`, `window2`, `window3`, and `window4`, where each member is a structure. At any given time, each union element holds one of the four possible structure members. Thus, the `screen` variable is a composite of up to four different "windows."

4.4.5 Array Declarations

Syntax

type-specifier declarator [*constant-expression*];
type-specifier declarator [ ];

A declaration for an array defines the name of the array and the type of each element. It can also define the number of elements in the array. A variable with array type is considered a pointer to the type of the array elements, as described in Section 5.2.2, "Identifiers."

Array declarations have two forms, as shown above. The *declarator* gives the variable name, and may modify the variable's type. The brackets ([ ]) following the *declarator* modify the declarator to array type. The *constant-expression* inside the brackets defines the number of elements in the array. Each element has the type given by the *type-specifier*, which can specify any type except **void** and function types.

The second form omits the *constant-expression* in brackets. This form can be used only if the array is initialized, declared as a formal parameter, or declared as a reference to an array explicitly defined elsewhere in the program.

Arrays of arrays, or "multidimensional" arrays, are defined by giving a list of bracketed *constant-expressions* following the array declarator:

type-specifier declarator[*constant-expression*] [*constant-expression*] ...

Each *constant-expression* in brackets defines the number of elements in a given dimension: two-dimensional arrays have two bracketed expressions, three-dimensional arrays have three, and so on. When a multidimensional array is declared within a function, the first *constant-expression* can be omitted if the array is initialized, declared as a formal parameter, or declared as a reference to an array explicitly defined elsewhere in the program.

Arrays of pointers to various types can be defined by using complex declarators, as described earlier in Section 4.3.2.

The storage associated with an array type is the storage required for all of its elements. The elements of an array are stored in contiguous and increasing memory locations, from the first element to the last. No blanks occur between the elements of an array in storage.

Arrays are stored by row. For example, the following array consists of two rows with three columns each:

```
char A[2][3];
```

The three columns of the first row are stored first, followed by the three columns of the second row. This means that the last subscript varies most quickly.

To refer to an individual element of an array, use a subscript expression, discussed in Section 5.2.5.

Examples

```
/******************** Example 1 ********************/

int scores[10], game;

/******************** Example 2 ********************/

float matrix[10][15];

/******************** Example 3 ********************/

struct {
            float x,y;
            } complex[100];

/******************** Example 4 ********************/

char *name[20];
```

The first example defines an array variable named `scores` with 10 elements, each of which has **int** type. The variable named `game` is declared as a simple variable with **int** type.

The second example defines a two-dimensional array named `matrix`. The array has 150 elements, each having **float** type.

The third example defines an array of structures. This array has 100 elements; each element is a structure containing two members.

The fourth example defines an array of pointers. The array has 20 elements, each of which is a pointer to a **char** value.

4.4.6 Pointer Declarations

Syntax

type-specifier ***declarator***;

A pointer declaration defines the name of the pointer variable and the type of the object to which the variable points. The *declarator* defines the variable's name, and may modify its type. The *type-specifier* gives the type of the object, which can be any fundamental, structure, or union type.

Pointer variables can also point to functions, arrays, and other pointers. For information on declaring more complex pointer types, refer to Section 4.3.2, "Complex Declarators."

A pointer to a structure or union type can be declared before the structure or union type is defined, as long as the pointer is not used before the type is defined. Such declarations are allowed because the compiler does not need to know the size of the structure or union to allocate space for the pointer variable. The pointer is declared by using the structure or union tag (see the fourth example below).

A variable declared as a pointer holds a memory address. The amount of storage required for an address and the meaning of the address depend on the given implementation of the compiler. Pointers to different types are not guaranteed to have the same length.

In some implementations, the special keywords **near**, **far**, and **huge** are available to modify the size of a pointer. Declarations using special keywords are described in Section 4.3.3. See your system documentation for more information on the meaning and use of these keywords.

Examples

```
char *message;                          /* Example 1 */

int *pointers[10]                       /* Example 2 */

int (*pointer)[10];                     /* Example 3 */

struct list *next, *previous;           /* Example 4 */

struct list {                           /* Example 5 */
            char *token;
            int count;
```

```
            struct list *next;
    } line;

struct id {                              /* Example 6 */
        unsigned int id_no;
        struct name *pname;
    } record;
```

The first example defines a pointer variable named `message`. It points to a variable with **char** type.

The second example defines an array of pointers named `pointers`. The array has 10 elements; each element is a pointer to a variable with **int** type.

The third example defines a pointer variable named `pointer`; it points to an array with 10 elements. Each element in this array has **int** type.

The fourth example defines two pointer variables that point to the structure type `list`. This declaration can appear before the definition of the `list` structure type (see the next example), as long as the `list` type definition has the same visibility as the declaration.

The fifth example declares the variable `line` to have the structure type named `list`. The `list` structure type is defined to have three members; the first member is a pointer to a **char** value, the second is an **int** value, and the third is a pointer to another `list` structure.

The sixth example declares the variable `record` to have the structure type `id`. Note that `pname` is declared as a pointer to another structure type `name`. This declaration can occur before the `name` type is defined.

4.5 Function Declarations

Syntax

[[*type-specifier*]] *declarator*([[*arg-type-list*]]) [[, *declarator...*]];

A function declaration defines the name and return type of a function, and possibly establishes the types and number of arguments to the function. Function declarations, also called forward declarations, do not define the function body or parameters; instead, they permit the attributes of the function to be known before the function is defined. Function definitions are described in detail in Section 7.2.

The *declarator* of the function declaration names the function, and the *type-specifier* gives the function's return type. If the *type-specifier* is omitted from a function declaration, the return type of the function is assumed to be **int**.

Function declarations may include either the **extern** or the **static** storage-class specifier. Storage-class specifiers are discussed in Section 4.6.

Argument-Type List

The *arg-type-list* establishes the number and types of the arguments to the function, and has the following form:

type-name-list[[,...]]

The *type-name-list* is a list of one or more type names. Each *type-name* is separated from the next by a comma. The first *type-name* gives the type of the first argument to the function, the second *type-name* gives the type of the second argument, and so on. If the *arg-type-list* ends with a comma followed by three periods (,...), the number of arguments to the function is variable. However, the function is expected to have at least as many arguments as there are *type-names* preceding the last comma.

If the *arg-type-list* contains only three periods (...), the number of arguments to the function is variable and may be zero.

Note

To maintain compatibility with previous versions, the compiler will also accept the comma character, without the trailing periods, at the end of the *arg-type-list* to indicate a variable number of arguments. A single comma can also be used instead of three periods to form the *arg-type-list* of a function taking zero or more arguments. Use of the comma is supported only for compatibility; using the three periods is recommended for new code.

A *type-name* for a fundamental, structure, or union type consists of the type specifier for that type (such as **int**). The *type-names* for pointers, arrays, and functions are formed by combining a type specifier

with an "abstract declarator"; that is, a declarator without an identifier. Section 4.9, "Type Names," explains how to form and interpret abstract declarators.

The special keyword **void** can be used in place of the *arg-type-list* to declare a function that has no arguments. The compiler produces a warning message if a call to the function or a call to the function definition specifies arguments.

One other special construction is allowed in the *arg-type-list*. The phrase **void** * specifies an argument of any pointer type. This phrase can be used in the *arg-type-list* as if it were a *type-name*.

The *arg-type-list* may be omitted. In this case the parentheses after the function identifier are still required, but they are empty. In this form the function declaration establishes neither the number nor the types of arguments to the function. When this information is omitted, the compiler does not perform any type checking between the actual arguments in a function call and the formal parameters of the function definition. See Section 7.4, "Function Calls," for more information.

Return Type

Functions can return values of any type except arrays and functions. Therefore, the *type-specifier* of a function declaration can specify any fundamental, structure, or union type. The function identifier can be modified with one or more asterisks (*) to declare a pointer return type.

Although functions are not allowed to return arrays and functions, they can return pointers to arrays and functions. Functions that return pointers to array or function types are declared by modifying the function identifier with asterisks (*), brackets ([ ]), and parentheses to form a complex declarator. Forming and interpreting complex declarators is discussed in Section 4.3.2.

Examples

```
int add(int, int);                    /* Example 1 */

double calc();                        /* Example 2 */

char *strfind(char *,...);            /* Example 3 */

void draw(void);                      /* Example 4 */
```

```
double (*sum(double, double))[3];    /* Example 5 */

int (*select(void))(int);            /* Example 6 */

char *p;                             /* Example 7 */
short *q;
int prt(void *);
```

The first example declares a function named `add` that takes two **int** arguments and returns an **int** value.

The second example declares a function named `calc` that returns a **double** value. No argument-type list is given.

The third example declares a function named `strfind`, which returns a pointer to a **char** value. The function takes at least one argument, a pointer to a **char** value. The argument-type list ends with a comma followed by three periods, indicating that the function may take more arguments.

The fourth example declares a function with **void** return type (returning no value). The *argument-type-list* is also **void**, meaning no arguments are expected for this function.

In the fifth example, `sum` is declared as a function returning a pointer to an array of three **double** values. The `sum` function takes two arguments, each a **double** value.

In the sixth example, the function named `select` is declared to take no arguments and return a pointer to a function. The pointer return value points to a function taking one **int** argument and returning an **int** value.

In the seventh example, the function `prt` is declared to take a pointer argument of any type and return an **int**. Either the **char** pointer `p` or the **short** pointer `q` could be passed as an argument to `prt` without producing a type-mismatch warning.

4.6 Storage Classes

The storage class of a variable determines whether the item has a "global" or "local" lifetime. An item with a global lifetime exists and has a value throughout the duration of the program. All functions have global lifetimes.

Variables with local lifetimes are allocated new storage each time execution control passes to the block in which they are defined. When execution control passes out of the block, the variables no longer have meaningful values.

Although C defines only two types of storage classes, four storage-class specifiers are available. They are as follows:

auto
register
static
extern

Items with **auto** and **register** class have local lifetimes. The **static** and **extern** specifiers refer to items with global lifetimes.

The four storage-class specifiers have distinct meanings because storage-class specifiers affect the visibility of functions and variables, as well as their storage class. The term "visibility" refers to the portion of the source program in which the variable or function can be referenced. An item with a global lifetime exists throughout the execution of the source program, but it may not be "visible" in all parts of the program. Visibility and the related concept of lifetime are discussed in Section 3.5.

The placement of variable or function declarations within source files also affects storage class and visibility. Declarations outside all function definitions are said to occur at the "external level"; declarations within function definitions occur at the "internal level."

The exact meaning of each storage-class specifier depends on whether the declaration occurs at the external or internal level and whether the item declared is a variable or a function. The following sections describe the meaning of storage-class specifiers in each kind of declaration; they also explain the default behavior when the storage-class specifier is omitted from a variable or function declaration.

4.6.1 Variable Declarations at the External Level

Variable declarations at the external level use the **static** and **extern** storage-class specifiers or omit the storage-class specifier entirely. The **auto** and **register** storage-class specifiers are not allowed at the external level.

Variable declarations at the external level are either *definitions* of variables or *references* to variables defined elsewhere. An external variable declaration that also initializes the variable (implicitly or explicitly) is a definition of the variable. Definitions at the external level can take several forms:

- A variable can be defined at the external level by declaring it with the **static** storage-class specifier. The **static** variable can be explicitly initialized with a constant expression, as described in Section 4.7. If the initializer is omitted, the variable is automatically initialized to zero at compile time. Thus, `static int k = 16;` and `static int k;` are both considered definitions.
- A variable is defined when it is explicitly initialized at the external level. For example, `int j = 3;` is a variable definition.

Once a variable is defined at the external level, it is visible throughout the remainder of the source file in which it appears. The variable is not visible above its definition in the same source file, nor is it visible in other source files of the program, unless a *reference* is declared to make it visible, as described below.

A variable can be defined at the external level only once within a source file. If the **static** storage-class specifier is given, another variable with the same name can be defined with the **static** storage-class specifier in a different source file. Since each **static** definition is visible only in its own source file, no conflict occurs.

The **extern** storage-class specifier is used to declare a *reference* to a variable defined elsewhere. These declarations can be used to make a definition in another source file visible or to make a variable visible above its definition in the same source file. Once a reference to the variable is declared at the external level, the variable is visible throughout the remainder of the source file in which the declared reference occurs.

Declarations that use the **extern** storage-class specifier are not allowed to contain initializers, since they refer to variables whose values are already defined.

For an **extern** reference to be valid, the variable to which it refers must be defined once, and only once, at the external level. The definition can be in any of the source files that form the program.

One special case is not covered by the rules outlined above. You can omit both the storage-class specifier and the initializer from a variable declaration at the external level; for example, the declaration `int n;` is a valid external declaration. This declaration can have one of two different meanings, depending on the context:

1. If a variable by the same name is *defined* at the external level elsewhere in the program, the declaration is taken to be a reference to that variable, exactly as if the **extern** storage-class specifier had been used in the declaration.
2. If no such definition is present, the declared variable is allocated storage at link time and initialized to 0. If more than one such declaration appears in the program, storage is allocated for the largest size declared for the variable. For example, if a program contains two uninitialized declarations of `i` at the external level, `int i;` and `char i;`, storage space for an **int** is allocated for `i` at link time.

Uninitialized variable declarations at the external level are not recommended for any file that might be placed in a library.

Example

```
/*********************************************************
                SOURCE FILE ONE
*********************************************************/

extern int i;                   /* reference to i,
                                   defined below */

main()
{
        i++;
        printf("%d\n", i);      /* i equals 4 */
        next();
}

int i = 3;                      /* definition of i */

next()
{
        i++;
        printf("%d\n", i);      /* i equals 5 */
        other();
}
```

```
/*********************************************************
                SOURCE FILE TWO
*********************************************************/

extern int i;                   /* reference to i in
                                   first source file */

other()
{
        i++;
        printf("%d\n", i);      /* i equals 6 */
}
```

The two source files contain a total of three external declarations of `i`. Only one declaration contains an initialization; that declaration, `int i = 3;` , defines the global variable `i` with initial value 3. The **extern** declaration of `i` at the top of the first source file makes the global variable visible above its definition in the file. Without the **extern** declaration, the `main` function could not reference the global variable `i`. The **extern** declaration of `i` in the second source file makes the global variable visible in that source file.

All three functions perform the same task: they increase `i` and print it. (Assume that the `printf` function is defined elsewhere in the program.) The values printed are 4, 5, and 6.

If the variable `i` had not been initialized, it would have been automatically set to 0 at link time. The values printed in this case would be 1, 2, and 3.

4.6.2 Variable Declarations at the Internal Level

Any of the four storage-class specifiers can be used for variable declarations at the internal level. When the storage-class specifier is omitted from a variable declaration at the internal level, the default storage class is **auto**.

The **auto** storage-class specifier declares a variable with a local lifetime. The variable is visible only in the block in which it is declared. Declarations of **auto** variables can include initializers, as discussed later in this chapter. Variables with **auto** storage class are not initialized automatically, so they should be explicitly initialized when declared or assigned initial values in statements within the block. If not initialized, the values of **auto** variables are undefined.

The **register** storage-class specifier tells the compiler to give the variable storage in a register, if possible. Register storage usually results in faster access time and smaller code size. Variables declared with **register** storage class have the same visibility as **auto** variables.

The number of registers that can be used for variable storage is machine dependent. If no registers are available when the compiler encounters the **register** declaration, the variable is given **auto** storage class and stored in memory. The compiler assigns register storage to variables in exactly the same order in which the declarations appear in the source file. Register storage, if available, is only guaranteed for **int** and pointer types.

A variable declared at the internal level with the **static** storage-class specifier has a global lifetime, and is visible only within the block in which it is declared. Unlike **auto** variables, variables declared as **static** retain their values when the block is exited.

A **static** variable can be initialized with a constant expression. If not explicitly initialized, a **static** variable is automatically set to 0. Initialization is performed once, at compile time. The **static** variable is *not* reinitialized each time the block is entered.

A variable declared with the **extern** storage-class specifier is a reference to a variable with the same name defined at the external level in any of the source files of the program. The purpose of the internal **extern** declaration is to make the external-level variable definition visible within the block. The internal **extern** declaration does not change the visibility of the global variable in any other part of the program.

Example

```
int i = 1;

main()
{       /* reference to i, defined above */
        extern int i;

        /* initial value is zero; a is
           visible only within main */
        static int a;

        /* b is stored in a register, if possible */
        register int b = 0;
```

```
        /* default storage class is auto */
        int c = 0;

        /* values printed are 1, 0, 0, 0 */
        printf("%d\n%d\n%d\n%d\n", i, a, b, c);
        other();
}

other()
{
        /* i is redefined */
        int i = 16;

        /* this a is visible only within other */
        static int a = 2;

        a += 2;
        /* values printed are 16, 4 */
        printf("%d\n%d\n", i, a);
}
```

The variable `i` is defined at the external level with initial value 1. A reference to the external-level `i` is declared in the `main` function with an **extern** declaration. The **static** variable `a` is automatically set to 0, since the initializer is omitted. The call to `printf` (assuming the `printf` function is defined elsewhere in the source program) prints out the values 1, 0, 0, 0.

In the `other` function, the variable `i` is redefined as a local variable with initial value 16. This does not affect the value of the external-level `i`. The variable `a` is declared as a **static** variable and initialized to 2. This `a` does not conflict with the `a` in `main`, since the visibility of **static** variables at the internal level is restricted to the block in which they are declared.

The variable `a` is increased by 2, giving 4 as the result. If the `other` function were called again in the same program, the initial value of `a` would be 4. Internal **static** variables retain their values when the block in which they are declared is exited and reentered.

4.6.3 Function Declarations at the External and Internal Levels

Function declarations can use either the **static** or the **extern** storage-class specifier. Functions always have global lifetimes.

The visibility rules for functions are slightly different from the rules for variables. Function declarations at the internal level have the same meaning as function declarations at the external level. This means that functions cannot have block visibility, and the visibility of functions cannot be nested. A function declared to be **static** is visible only within the source file in which it is defined. Any function in the same source file can call the **static** function, but functions in other source files cannot. Another **static** function by the same name can be declared in a different source file without conflict.

Functions declared as **extern** are visible throughout all the source files that constitute the program (unless they are later redeclared as **static**). Any function can call an **extern** function.

Function declarations that omit the storage-class specifier default to **extern**.

4.7 Initialization

A variable can be set to an initial value by applying an initializer to the declarator in the variable declaration. The value or values of the initializer are assigned to the variable. The initializer is preceded by an equal sign (=), as shown below:

= *initializer*

Variables of any type can be initialized, with the restrictions outlined below. Functions do not take initializers.

Declarations that use the **extern** storage-class specifier cannot contain initializers.

Variables declared at the external level can be initialized; if not explicitly initialized, they are set to 0 at compile time. Any variable declared with the **static** storage-class specifier can be initialized with a constant expression. Initializations of **static** variables are performed once, at compile time. If not explicitly initialized, **static** variables are automatically set to 0.

Initializations of **auto** and **register** variables are performed each time execution control passes to the block in which they are declared. If the initializer is omitted from the declaration of an **auto** or **register** variable, the initial value of the variable is undefined.

Initializations of **auto** aggregate types (arrays, structures, and unions) are prohibited. Only **static** aggregates and aggregates declared at the external level can be initialized.

The initial values for external variable declarations and for all **static** variables, whether external or internal, must be constant expressions. Constant expressions are described in Section 5.2.10. Automatic and register variables can be initialized with either constant or variable values.

Sections 4.7.1 and 4.7.2 describe how to initialize variables of fundamental, pointer, and aggregate types.

4.7.1 Fundamental and Pointer Types

Syntax

= *expression*

The value of *expression* is assigned to the variable. The conversion rules for assignment apply.

Examples

```
int x = 10;                 /* Example 1 */

register int *px = 0;       /* Example 2 */

int c = (3 * 1024);         /* Example 3 */

int *b = &x;                /* Example 4 */
```

In the first example, `x` is initialized to the constant expression 10. In the second example, the pointer `px` is initialized to 0, producing a "null" pointer. The third example uses a constant expression to initialize `c`. The fourth example initializes the pointer `b` with the address of another variable, `x`.

4.7.2 Aggregate Types

Syntax

= { *initializer-list*}

An *initializer-list* is a list of initializers separated by commas. Each initializer in the list is either a constant expression or an *initializer-list.* Therefore, a brace-enclosed list can appear within another *initializer-list.* This is useful for initializing aggregate members of an aggregate, as shown in the examples below.

For each *initializer-list,* the values of the constant expressions are assigned in order to the members of the aggregate variable. When a union is initialized, the *initializer-list* must be a single constant expression. The value of the constant expression is assigned to the first member of the union.

If there are fewer values in an *initializer-list* than there are in the aggregate type, the remaining members or elements are initialized to 0. Giving too many initial values for the aggregate type causes an error. These rules apply to each embedded *initializer-list,* as well as to the aggregate as a whole.

For example,

```
int P[4][3] = {
        { 1, 1, 1 },
        { 2, 2, 2 },
        { 3, 3, 3,},
        { 4, 4, 4,},
};
```

declares `P` as a 4-by-3 array and initializes the elements of its first row to 1, the elements of its second row to 2, and so on through the fourth row. Note that the *initializer-list* for the third and fourth rows contains commas after the last constant expression. The last *initializer-list* (`{4, 4, 4,}`) is also followed by a comma. These extra commas are permitted but are not required; the required commas are those that separate constant expressions and *initializer-lists.*

If there is no embedded initializer list for an aggregate member, values are simply assigned, in order, to each member of the subaggregate. Therefore, the above initialization is equivalent to the following:

```
int P[4][3] = {
        1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4
};
```

Braces can also appear around individual initializers in the list.

When initializing aggregate variables, care must be taken to use braces and initializer lists properly. The following example illustrates in more detail the compiler's interpretation of braces:

```
typedef struct {
    int n1, n2, n3;
} triplet;

triplet nlist[2][3] = {
    { {  1,  2,  3 }, {  4,  5,  6 }, {  7,  8,  9 } },   /* Line 1 */
    { { 10,11,12 }, { 13,14,15 }, { 15,16,17 } }   /* Line 2 */
};
```

In the above example, `nlist` is declared as a 2-by-3 array of structures, each structure having three members. Line 1 of the initialization assigns values to the first row of `nlist`, as follows:

1. The first left brace on Line 1 signals the compiler that initialization of the first aggregate member of `nlist` is beginning (that is, `nlist[0]`).
2. The second left brace indicates that initialization of the first aggregate member of `nlist[0]` is beginning (that is, the structure at `nlist[0][0]`).
3. The first right brace ends initialization of the structure `nlist[0][0]`; the next left brace starts initialization of `nlist[0][1]`.
4. The process continues to the end of the line, where the closing right brace ends initialization of `nlist[0]`.

Similarly, Line 2 assigns values to the second row of `nlist`.

Note that the outer sets of braces enclosing the initializers on Line 1 and on Line 2 are required. The following construction, which omits the outer braces, would cause an error:

```
        /* THIS CAUSES AN ERROR */
triplet nlist[2][3] = {
        {  1,  2,  3 },{  4,  5,  6 },{  7,  8,  9 },    /* Line 1 */
        { 10,11,12 },{ 13,14,15 },{ 16,17,18 }    /* Line 2 */
};
```

In the above construction, the first left brace on Line 1 starts the initialization of `nlist[0]`, which is an array of three structures. The values 1, 2, and 3 are assigned to the three members of the first structure. When the next right brace is encountered (after the value 3), initialization of `nlist[0]` is complete, and the two remaining structures in the three-structure array are automatically initialized to 0. Similarly, `{ 4,5,6 }` initializes the first structure in the second row of `nlist`, and the remaining two structures of `nlist[1]` are set to 0. When the compiler encounters the next initializer list (`{ 7,8,9 }`), it attempts to initialize `nlist[2]`. Since `nlist` has only two rows, this produces an error.

Examples

```
/******************** Example 1 ********************/

struct list {
        int i, j, k;
        float m[2][3];
        } x = {
                1,
                2,
                3,
                {4.0, 4.0, 4.0}
        };

/******************** Example 2 ********************/

union {
        char x[2][3];
        int i, j, k;
        } y = {
                {'1'},
                {'4'}
        };
```

In the first example, the three **int** members of `x` are initialized to 1, 2, and 3, respectively. The three elements in the first row of `m` are initialized to 4.0; the elements of the remaining row of `m` are initialized to 0 by default.

In the second example, the union variable y is initialized. The first element of the union is an array, so the initializer is an aggregate initializer. The initializer list { '1' } gives values to the first row of the array. Since only one value appears in the list, the element in the first column is initialized to the character 1, and the remaining two elements in the row are initialized to 0 (the null character), by default. Similarly, the first element of the second row of x is initialized to the character 4, and the remaining two elements in the row are initialized to 0.

4.7.3 String Initializers

An array can be initialized with a string literal. For example,

```
char code[ ] = "abc";
```

initializes code as a four-element array of characters. The fourth element is the null character that terminates all string literals.

If the array size is specified and the string is longer than the specified size of the array, the extra characters are simply discarded. The following declaration initializes code as a three-element character array:

```
char code[3] = "abcd";
```

Only the first three characters of the initializer are assigned to code. The character d and the null character are discarded.

If the string is shorter than the specified size of the array, the remaining elements of the array are initialized to 0 (the null character).

4.8 Type Declarations

A type declaration defines the name and members of a structure or union type, or the name and enumeration set of an enumeration type. The name of a declared type can be used in variable or function declarations to refer to that type. This is useful if many variables and functions have the same type.

A **typedef** declaration defines a type specifier for a type. These declarations are used to construct shorter or more meaningful names for types already defined by C or for types declared by the user.

4.8.1 Structure, Union, and Enumeration Types

Declarations of structure, union, and enumeration types have the same general form as variable declarations of those types. In type declarations the variable identifier is omitted, since no variable is declared. The *tag* is mandatory; it names the structure, union, or enumeration type. The *member-declaration-list* or *enum-list* defining the type must appear in the type declaration; the abbreviated form of variable declarations, in which a *tag* refers to a type defined elsewhere, is not legal for type declarations.

Examples

```
/******************** Example 1 ********************/

enum status {
        loss = -1,
        bye,
        tie = 0,
        win
        };

/******************** Example 2 ********************/

struct student {
        char name[20];
        int id, class;
        };
```

The first example declares an enumeration type named `status`. The name of the type can be used in declarations of enumeration variables. The identifier `loss` is explicitly set to -1. Both `bye` and `tie` are associated with the value 0, and `win` is given the value 1.

The second example declares a structure type named `student`. A declaration such as `struct student employee;` can be used to declare a structure variable with `student` type.

4.8.2 Typedef Declarations

Syntax

typedef *type-specifier declarator* [[, *declarator*...]];

A **typedef** declaration is analogous to a variable declaration except that the **typedef** keyword replaces a storage-class specifier. The declaration is interpreted in the same way as variable and function declarations, but the identifier, instead of assuming the type specified by the declaration, becomes a synonym for the type.

A **typedef** declaration does not create types. It creates synonyms for existing types, or names for types that could be specified in other ways. Any type can be declared with **typedef**, including pointer, function, and array types. A **typedef** name for a pointer to a structure or union type can be declared before the structure or union type is defined, as long as the definition has the same visibility as the declaration.

Examples

```
/******************** Example 1 ********************/

typedef int WHOLE;

/******************** Example 2 ********************/

typedef struct club {
        char name[30];
        int size, year;
        } GROUP ;

/******************** Example 3 ********************/

typedef GROUP *PG;

/******************** Example 4 ********************/

typedef void DRAWF(int, int);
```

The first example declares WHOLE to be a synonym for **int**.

The second example declares GROUP as a structure type with three members. Since a structure tag, `club`, is also specified, either the **typedef** name (GROUP) or the structure tag can be used in declarations.

The third example uses the previous **typedef** name to declare a pointer type. The type PG is declared as a pointer to the GROUP type, which in turn is defined as a structure type.

The final example provides the type DRAWF for a function returning no value and taking two **int** arguments. This means, for example, that the declaration DRAWF `box;` is equivalent to the declaration `void box(int, int);`.

4.9 Type Names

A "type name" specifies a particular data type. Type names are used in three contexts: in the argument-type lists of function declarations, in type casts, and in **sizeof** operations. Argument-type lists are discussed in Section 4.5, "Function Declarations." Type casts and **sizeof** operations are discussed in sections 5.7.2 and 5.3.4, respectively.

The type names for fundamental, enumeration, structure, and union types are simply the type specifiers for those types.

A type name for a pointer, array, or function type has the following form:

type-specifier abstract-declarator

An *abstract-declarator* is a declarator without an identifier, consisting of one or more pointer, array, or function modifiers. The pointer modifier (*) always appears before the identifier in a declarator, while array ([ ]) and function (( )) modifiers appear after the identifier. It is thus possible to determine where the identifier would appear in an abstract declarator and interpret the declarator accordingly.

Abstract declarators can be complex. Parentheses in a complex abstract declarator specify a particular interpretation, just as they do for the complex declarators in declarations.

The abstract declarator consisting of a set of empty parentheses, ( ), is not allowed because it is ambiguous. It is impossible to determine whether the

implied identifier belongs inside the parentheses, in which case it is an unmodified type, or before the parentheses, in which case it is a function type.

The type specifiers established through **typedef** declarations also qualify as type names.

Examples

```
long *                          /* Example 1 */

int (*)[5]                      /* Example 2 */

int (*)(void)                   /* Example 3 */
```

The first example gives the type name for "pointer to **long**" type.

The second and third examples show how parentheses modify complex abstract declarators. Example 2 gives the type name for a pointer to an array of five **int** values. Example 3 names a pointer to a function taking no arguments and returning an **int**.

Chapter 5
Expressions and Assignments

5.1 Introduction

This chapter describes how to form expressions and make assignments in the C language. An expression is a combination of operands and operators that yields ("expresses") a single value. An operand is a constant or variable value that is manipulated in the expression. Each operand of an expression is also an expression, since it represents a single value. Operators specify how the operand or operands of the expression are manipulated.

In C, assignments are considered expressions. An assignment yields a value; its value is the value being assigned. In addition to the simple assignment operator (=), C offers complex assignment operators that both transform and assign their operands.

The value resulting from an expression's evaluation depends on the relative precedence of operators in the expression and on side effects, if present. The precedence of operators determines the grouping of operands in an expression. Side effects are changes caused by the evaluation of an expression. In an expression with side effects, the evaluation of one operand can affect the value of another. With some operators, the order in which operands are evaluated also affects the result of the expression.

The value represented by each operand in an expression has a type, which may be converted to a different type in certain contexts. Type conversions occur in assignments, type casts, function calls, and operations.

5.2 Operands

A C operand is a constant, an identifier, a string, a function call, a subscript expression, a member-selection expression, or a more complex expression formed by combining operands with operators or enclosing operands in parentheses. Any operand that yields a constant value is called a "constant expression."

Every operand has a type. The following sections discuss the type of value each kind of operand represents. An operand can be cast from its original type to another type by means of a "type-cast" operation. A type-cast expression can also form an operand of an expression.

5.2.1 Constants

A constant operand has the value and type of the constant value it represents. A character constant has **int** type. An integer constant has **int**, **long**, **unsigned int**, or **unsigned long** type, depending on the integer's size and how the value is specified. Floating-point constants always have **double** type. String literals are considered arrays of characters and are discussed in Section 5.2.3.

5.2.2 Identifiers

An identifier names a variable or function. Every identifier has a type, which is established when the identifier is declared. The value of an identifier depends on its type, as follows:

- Identifiers of integral and floating-point types represent values of the corresponding type.

- An identifier of **enum** type represents one constant value of a set of constant values. The value of the identifier is the constant value. Its type is **int**, by definition of the **enum** type.

- An identifier of **struct** or **union** type represents a value of the specified **struct** or **union** type.

- An identifier declared as a pointer represents a pointer to the specified type.

- An identifier declared as an array represents a pointer whose value is the address of the first element of the array. The type addressed by the pointer is the type of the elements of the array. For example, if `series` is declared to be a 10-element integer array, the identifier `series` expresses the address of the array, while the subscript expression `series[5]` refers to a variable integer element of `series`. Subscript expressions are discussed in Section 5.2.5.

  The address of an array does not change during the execution of the program, although the values of the individual elements can change. The pointer value represented by an array identifier is not a variable, and an array identifier cannot form the left-hand operand of an assignment operation.

- An identifier declared as a function represents a pointer whose value is the address of the function. The type addressed by the pointer is a function returning a value of a specified type. The address of a

function does not change during the execution of a program; only the return value varies. Thus, function identifiers cannot be left-hand operands in assignment operations.

5.2.3 Strings

A string literal consists of a list of characters enclosed in double quotes, as shown below:

"*string*"

A string literal is stored as an array of elements with **char** type. The string literal represents the address of the first element of the array. The address of the string's first element is a constant, so the value represented by a string expression is a constant.

Since string literals are effectively pointers, they can be used in contexts that allow pointer values, and they are subject to the same restrictions as pointers. String literals have one additional restriction: they are not variables and cannot be left-hand operands in assignment operations.

The last character of a string is always the null character, **0**. The null character is not visible in the string expression, but it is added as the last element when the string is stored. Therefore, the string `"abc"` actually has four characters rather than three.

5.2.4 Function Calls

Syntax

expression (*expression-list*)

A function call consists of an *expression* followed by an *expression-list* in parentheses, where *expression* evaluates to a function address (for example, a function identifier), and *expression-list* is a list of expressions (separated by commas) whose values, the actual arguments, are passed to the function. The *expression-list* can be empty.

A function-call expression has the value and type of the function's return value. If the function's return type is **void**, the function-call expression also has **void** type. If control returns from the called function without execution of a **return** statement, the value of the function call is undefined. See Section 7.4 for more information about function calls.

5.2.5 Subscript Expressions

Syntax

expression1 [*expression2*]

A subscript expression represents the value at the address that is *expression2* positions beyond *expression1*. *Expression1* is any pointer value, such as an array identifier, and *expression2* is an integral value. *Expression2* must be enclosed in brackets ([ ]).

Subscript expressions are generally used to refer to array elements, but a subscript can be applied to any pointer.

The subscript expression is evaluated by adding the integral value (*expression2*) to the pointer value (*expression1*), then applying the indirection operator (*) to the result. (See Section 5.3.3 for a discussion of the indirection operator.) In effect, for a one-dimensional array, the following four expressions are equivalent, assuming that `a` is a pointer and `b` is an integer.

```
a[b]
*(a + b)
*(b + a)
b[a]
```

According to the conversion rules of the addition operator (see Section 5.3.6), the integral value is converted to an address offset by multiplying it by the length of the type addressed by the pointer.

For example, suppose the identifier `line` refers to an array of **int** values. To evaluate the expression `line[i]`, the integer value `i` is multiplied by the length of an **int**. The converted value of `i` represents `i` **int** positions. This converted value is added to the original pointer value (`line`) to yield an address that is offset `i` **int** positions from `line`.

As the last step in evaluating the subscript expression, the indirection operator is applied to the new address. The result is the value of the array element at that position (intuitively, `line[i]`).

Note that the subscript expression

```
line[0]
```

represents the value of the first element of `line`, since the offset from the

address represented by `line` is 0. Similarly, an expression such as

```
line[5]
```

refers to the element offset five positions from `line`, or the sixth element of the array.

Multidimensional-Array References

A subscript expression can be subscripted, as follows:

expression1 [*expression2*] [*expression3*]...

Subscript expressions associate left to right. The leftmost subscript expression, *expression1*[*expression2*], is evaluated first. The address that results from adding *expression1* and *expression2* forms the pointer expression to which *expression3* is added. The indirection operator (*) is applied after the last subscripted expression is evaluated. However, the indirection operator is not applied if the final pointer value addresses an array type (see the third example below).

Expressions with multiple subscripts refer to elements of multidimensional arrays. A multidimensional array is an array whose elements are arrays. The first element of a three-dimensional array, for example, is an array with two dimensions.

Examples

```
int prop[3][4][6];
int i, *ip, (*ipp)[6];

i = prop[0][0][1];              /* Example 1 */

i = prop[2][1][3];              /* Example 2 */

ip = prop[2][1];                /* Example 3 */

ipp = prop[2];                  /* Example 4 */
```

The array named `prop` has 3 elements, each of which is a 4-by-6 array of **int** values.

Example 1 shows how to refer to the second individual **int** element of `prop`. Arrays are stored by row, so the last subscript varies the most quickly; the expression `prop[0][0][2]` refers to the next (third) element in the array, and so on.

The second example shows a more complex reference to an individual element of `prop`. The expression is evaluated as follows:

1. The first subscript, `2`, is multiplied by the size of a 4-by-6 **int** array and added to the pointer value `prop`. The result points to the third 4-by-6 array of `prop`.
2. Next, the second subscript, `1`, is multiplied by the size of the 6-element **int** array and added to the address represented by `prop[2]`.
3. Each element of the 6-element array is an **int** value, so the final subscript, `3`, is multiplied by the size of an **int** before it is added to `prop[2][1]`. The resulting pointer addresses the fourth element of the 6-element array.
4. The last step in evaluating the expression `prop[2][1][3]` is applying the indirection operator to the pointer value. The result is the **int** element at that address.

Examples 3 and 4 show cases where the indirection operator is not applied. In Example 3, the expression `prop[2][1]` is a valid reference to the three-dimensional array `prop`; it refers to a 6-element array. Since the pointer value addresses an array, the indirection operator is not applied. Similarly, the result of the expression `prop[2]` in Example 4 is a pointer value addressing an array with two dimensions.

5.2.6 Member-Selection Expressions

Syntax

expression.identifier
expression->identifier

Member-selection expressions refer to members of structures and unions. A member-selection expression has the value and type of the selected member.

In the first form, *expression.identifier*, *expression* represents a value of **struct** or **union** type, and the *identifier* names a member of the specified structure or union.

In the second form, *expression*–>*identifier*, *expression* represents a pointer to a structure or union, and the *identifier* names a member of the specified structure or union.

The two forms of member-selection expressions have similar effects. In fact, expressions involving the pointer selection operator (–>) are shorthand versions of expressions using the period (.) in cases in which the expression before the period consists of the indirection operator (*) applied to a pointer value. (The indirection operator is discussed in Section 5.3.3.) Therefore,

expression–>*identifier*

is equivalent to

(**expression*).*identifier*

when *expression* is a pointer value.

Examples

```
struct pair {
        int a;
        int b;
        struct pair *sp;
        } item, list[10];

item.sp = &item;                    /* Example 1 */

(item.sp)->a = 24;                  /* Example 2 */

list[8].b = 12;                     /* Example 3 */
```

In the first example, the address of the `item` structure is assigned to the `sp` member of the structure. This means that `item` contains a pointer to itself.

In the second example, the pointer expression `item.sp` is used with the pointer selection operator (–>) to assign a value to the member `a`.

The third example shows how to select an individual structure member from an array of structures.

5.2.7 Expressions with Operators

Expressions with operators can be unary, binary, or ternary expressions. A unary expression consists of an operand prefixed by a unary operator ("unop") or an operand enclosed in parentheses and preceded by the **sizeof** keyword:

unop operand
sizeof (*operand*)

A binary expression consists of two operands joined by a binary operator ("binop"):

operand binop operand

A ternary expression consists of three operands joined by the ternary (**?** **:**) operator:

operand **?** *operand* **:** *operand*

Assignment expressions use unary or binary assignment operators. The unary assignment operators are the increment (++) and decrement (--) operators; the binary assignment operators are the simple assignment operator (=) and the compound assignment operators (referred to as "compound-assign-ops"). Each compound assignment operator is a combination of another binary operator with the simple assignment operator. The forms of assignment expressions are as follows:

operand++
operand--
++*operand*
--*operand*
operand = *operand*
operand compound-assignment-op operand

5.2.8 Expressions in Parentheses

Any operand can be enclosed in parentheses; they have no effect on the type or value of the enclosed expression. For example, in the expression

```
(10 + 5) / 5
```

the parentheses around `10 + 5` mean that the value of `10 + 5` is the left operand of the division (/) operator. The result of `(10 + 5) / 5` is 3. Without the parentheses, `10 + 5 / 5` would evaluate to 11.

Although parentheses affect the way operands are grouped in an expression, they cannot guarantee a particular order of evaluation for the expression.

5.2.9 Type-Cast Expressions

A type-cast expression has the following form:

(*type-name*) *operand*

Type-cast conversions are discussed in Section 5.7.2; type names are discussed in Section 4.9.

5.2.10 Constant Expressions

A constant expression is any expression that evaluates to a constant. The operands of a constant expression can be integer constants, character constants, floating-point constants, enumeration constants, type casts to integral and floating-point types, and other constant expressions. The operands can be combined and modified using operators, as described in Section 5.2.7, with some restrictions.

Constant expressions cannot use assignment operators (see Section 5.4) or the binary sequential evaluation operator (,). The unary address-of operator (**&**) can be used only in certain initializations (see the last paragraph of this section, 5.2.10).

Constant expressions used in preprocessor directives are subject to additional restrictions, and are consequently known as *restricted-constant-expressions*. A *restricted-constant-expression* cannot contain **sizeof** expressions, enumeration constants, or type casts to any type. It can, however, contain the special constant expression **defined(***identifier***)**. See Section 8.2.1, "The #define Directive," for more information.

These additional restrictions also apply to constant expressions used to initialize variables at the external level. However, such expressions are allowed to apply the unary address-of operator (&) to other external-level variables with fundamental, structure, and union types and to external-level arrays subscripted with a constant expression. In these expressions, a constant expression not involving the address-of operator can be added to or subtracted from the address expression.

5.3 Operators

C operators take one operand (unary operators), two operands (binary operators), or three operands (the ternary operator). Assignment operators are unary or binary operators; the assignment operators are described in Section 5.4.

Unary operators prefix their operand and associate right to left. C's unary operators are as follows:

Symbol	Name
– ~ !	Complement operators
* &	Indirection and address-of operators
sizeof	Size operator

Binary operators associate left to right. The binary operators are as follows:

Symbol	Name
* / %	Multiplicative operators
+ –	Additive operators
<< >>	Shift operators
< > <= >= == !=	Relational operators
& | ^	Bitwise operators
&& ||	Logical operators
,	Sequential-evaluation operator

C has one ternary operator, the conditional operator (**? :**). It associates right to left.

5.3.1 Usual Arithmetic Conversions

Most C operators perform type conversions to bring the operands of an expression to a common type or to extend short values to the integer size used in machine operations. The conversions performed by C operators depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integral and floating-point types. These conversions are known as "arithmetic" conversions because they apply to the types of values ordinarily used in arithmetic.

The arithmetic conversions summarized below are called the "usual arithmetic conversions." The discussion of each operator in the following sections specifies whether or not the operator performs the usual arithmetic conversions. It also specifies the additional conversions, if any, the operator performs.

The specific path of each type of conversion is outlined in Section 5.7.

The usual arithmetic conversions proceed in the following order:

1. Any operands of **float** type are converted to **double** type.
2. If one operand has **double** type, the other operand is converted to **double**.
3. Any operands of **char** or **short** type are converted to **int**.
4. Any operands of **unsigned char** or **unsigned short** type are converted to **unsigned int** type.
5. If one operand is of type **unsigned long**, the other operand is converted to **unsigned long**.
6. If one operand is of type **long**, the other operand is converted to **long**.
7. If one operand is of type **unsigned int**, the other operand is converted to **unsigned int**.

5.3.2 Complement Operators

Arithmetic Negation (–)

The arithmetic-negation operator (–) produces the negative (two's complement) of its operand. The operand must be an integral or floating-point value. The usual arithmetic conversions are performed.

Bitwise Complement (~)

The bitwise-complement operator (~) produces the bitwise complement of its operand. The operand must be of integral type. The usual arithmetic conversions are performed; the result has the type of the operand after conversion.

Logical-NOT (!)

The logical-NOT operator (!) produces the value 0 if its operand is true (nonzero) and the value 1 if its operand is false (0). The result has **int** type. The operand must be an integral, floating-point, or pointer value.

Examples

```
/******************** Example 1 ********************/
short x = 987;
    x = -x;

/******************** Example 2 ********************/
unsigned short y = 0xaaaa;
    y = ~y;

/******************** Example 3 ********************/
if ( !(x < y));
```

In the first example, the new value of x is the negative of 987, or −987.

In the second example, the new value assigned to y is the one's complement of the unsigned value 0xaaaa, or 0x5555.

In the third example, if x is greater than or equal to y, the result of the expression is 1 (true). If x is less than y, the result is 0 (false).

5.3.3 Indirection and Address-of Operators

Indirection (*)

The indirection operator (*) accesses a value indirectly, through a pointer. The operand must be a pointer value. The result of the operation is the value to which the operand points. The result type is the type addressed by the pointer operand. If the pointer value is null, the result is unpredictable.

Address-of (&)

The address-of operator (&) takes the address of its operand. The operand can be any value that can appear as the left-hand value of an assignment operation. (Assignment operations are discussed in Section 5.4.) The result of the address operation is a pointer to the operand. The type addressed by the pointer is the type of the operand.

The address-of operator cannot be applied to a bit-field member of a structure, nor can it be applied to an identifier declared with the **register** storage-class specifier.

Examples

```
int *pa, x;
int a[20];
double d;

pa = &a[5];                 /* Example 1 */

x = *pa;                    /* Example 2 */

if (x == *&x)               /* Example 3 */
    printf("True\n");

d = *(double *)(&x);        /* Example 4 */
```

In the first example, the address-of operator (&) takes the address of the sixth element of the array `a`. The result is stored in the pointer variable `pa`.

The indirection operator (*) is used in the second example to access the **int** value at the address stored in `pa`. The value is assigned to the integer variable `x`.

In Example 3, the word `True` would be printed. This example demonstrates that the result of applying the indirection operator to the address of `x` is the same as `x`.

Example 4 shows a useful application of the rule shown in Example 3. The address of `x` is converted by a type cast to a pointer to a **double**; the indirection operator is then applied, and the result of the expression is a **double** value.

5.3.4 The sizeof Operator

The **sizeof** operator determines the amount of storage associated with an identifier or a type. A **sizeof** expression has the form

sizeof(*name*)

where *name* is either an identifier or a type name. The type name may not be **void**. The value of a **sizeof** expression is the amount of storage, in bytes, associated with the named identifier or type.

When the **sizeof** operator is applied to an array identifier, the result is the size of the entire array in bytes rather than the size of the pointer represented by the array identifier.

When the **sizeof** operator is applied to a structure or union type name, or to an identifier of structure or union type, the result is the actual size in bytes of the structure or union, which may include internal and trailing padding used to align the members of the structure or union on memory boundaries. Thus, the result may not correspond to the size calculated by adding up the storage requirements of the members.

Example

```
buffer = calloc(100, sizeof (int) );
```

With the **sizeof** operator you can avoid specifying machine-dependent data sizes in your program. The above example uses the **sizeof** operator to pass the size of an **int**, which varies across machines, as an argument to a function named `calloc`. The value returned by the function is stored in `buffer`.

5.3.5 Multiplicative Operators

The multiplicative operators perform multiplication (*), division (/), and remainder (%) operations. The operands of the remainder operator (%) must be integral; the multiplication (*) and division (/) operators take integral and floating-point operands. The types of the operands can be different. The multiplicative operators perform the usual arithmetic conversions on the operands. The type of the result is the type of the operands after conversion.

The conversions performed by the multiplicative operators make no provision for overflow or underflow conditions. Information is lost if the result of a multiplicative operation cannot be represented in the type of the operands after conversion.

Multiplication (*)

The multiplication operator (*) specifies that its two operands are to be multiplied.

Division (/)

The division operator (/) specifies that its first operand is to be divided by the second. When two integers are divided, the result, if not an integer, is truncated. If both operands are positive or unsigned, the result is truncated toward 0. The direction of truncation when either operand is negative may be either toward or away from 0, depending on the implementation. Division by 0 gives unpredictable results.

Remainder (%)

The result of the remainder operator (%) is the remainder when the first operand is divided by the second.

Examples

```
int i = 10, j = 3, n;
double x = 2.0, y;

y = x * i;              /* Example 1 */

n = i / j;              /* Example 2 */

n = i % j;              /* Example 3 */
```

In the first example, `x` is multiplied by `i` to give the value 20.0. The result has **double** type.

In the second example, 10 is divided by 3. The result is truncated toward 0, yielding the integer value 3.

In the third example, `n` is assigned the integer remainder 1 when 10 is divided by 3.

5.3.6 Additive Operators

The additive operators perform addition (+) and subtraction (–). The operands can be integral or floating-point values; some additive operations can also be performed on pointer values, as outlined under the discussion of each operator. The usual arithmetic conversions are performed on integral and floating-point operands. The type of the result is the type of the operands after conversion.

The conversions performed by the additive operators make no provision for overflow or underflow conditions. Information is lost if the result of an additive operation cannot be represented in the type of the operands after conversion.

Addition (+)

The addition operator (+) specifies addition of its two operands. The operands can have integral or floating-point types, as described previously, or one operand can be a pointer and the other an integer. When an integer is added to a pointer, the integer value (i) is converted by multiplying it by the length of the value addressed by the pointer. After conversion, the integer value represents i memory positions, where each position has the length specified by the pointer type. When the converted integer value is added to the pointer value, the result is a new pointer value expressing the address i positions from the original address. The new pointer value addresses the same type as the original pointer value.

Subtraction (–)

The subtraction operator (–) subtracts its second operand from the first. The operands can be integral or floating-point values, as described earlier. The subtraction operator also allows the subtraction of an integer from a pointer value and the subtraction of two pointer values.

When an integer value is subtracted from a pointer value, the same conversions occur as with addition of a pointer and integer. The subtraction operator converts the integer value with respect to the type addressed by the pointer value. The result is the memory address i positions before the original address, where i is the integer value and each position is the length of the type addressed by the pointer value. The new pointer points to the type addressed by the original pointer value.

Two pointer values can be subtracted if they point to the same type. The difference between the two pointers is converted to a signed integer value by dividing the difference by the length of the type the pointers address. The result represents the number of memory positions of that type between the two addresses. The result is only guaranteed to be meaningful for two elements of the same array, as discussed next.

Pointer Arithmetic

Additive operations involving a pointer and an integer generally give meaningful results only when the pointer operand addresses an array member and the integer value produces an offset within the bounds of the same array. The conversion of the integer value to an address offset assumes that only memory positions of the same size lie between the original address and the address plus offset.

The preceding assumption is valid for array members. An array is by definition a series of values of the same type; its elements reside in contiguous memory locations. Storage for any types except array elements is not guaranteed to be completely filled. That is, blanks can occur between memory positions, even positions of the same type. Adding to or subtracting from addresses referring to any values but array elements gives unpredictable results.

Similarly, the conversion involved in the subtraction of two pointer values assumes that only values of the same type, with no blanks, lie between the two addresses given by the operands.

Additive operations between pointer and integer values on machines with segmented architecture (such as the 8086/8088) may not be valid in some cases. See your system documentation for more information.

Examples

```
int i = 4, j;
float x[10];
float *px;

px = &x[4] + i;          /* Example 1 */

j = &x[i] - &x[i-2];     /* Example 2 */
```

In the first example, the integer operand `i` is added to the address of the fifth element of `x`. The value of `i` is multiplied by the length of a **float** and added to `&x[4]`. The resulting pointer value is the address of `x[8]`.

In the second example, the address of the third element of `x` (given by `x[i-2]`) is subtracted from the address of the fifth element of `x` (given by `x[i]`). The difference is divided by the length of a **float**; the result is the integer value 2.

5.3.7 Shift Operators

The shift operators shift their first operand left (<<) or right (>>) by the number of positions the second operand specifies. Both operands must be integral values. The usual arithmetic conversions are performed; the type of the result is the type of the left operand after conversion.

For leftward shifts, the vacated right bits are set to 0. In a rightward shift, the method of filling left bits depends on the type, after conversion, of the first operand. If the type is **unsigned**, vacated left bits will be set to 0. Otherwise, vacated left bits are filled with copies of the sign bit.

The result of a shift operation is undefined if the second operand is negative.

The conversions performed by the shift operators make no provision for overflow or underflow conditions. Information is lost if the result of a shift operation cannot be represented in the type of the first operand after conversion.

Example

```
unsigned int x, y, z;

x = 0x00aa;
y = 0x5500;

z = (x << 8) + (y >> 8);
```

In the above example, `x` is shifted left by eight positions and `y` is shifted right eight positions. The shifted values are added, giving `0xaa55`, and assigned to `z`.

5.3.8 Relational Operators

The binary relational operators test their first operand against the second to determine if the relationship specified by the operator holds true. The result of a relational expression is 1 if the tested relationship holds and 0 if it does not. The type of the result is **int**. The relational operators test the following relationships:

Operator	Relationship Tested
<	First operand less than second operand
>	First operand greater than second operand
<=	First operand less than or equal to second operand
>=	First operand greater than or equal to second operand
==	First operand equal to second operand
!=	First operand not equal to second operand

The operands can have integral, floating-point, or pointer type. The types of the operands can be different. The usual arithmetic conversions are performed on integral and floating-point operands.

One or both operands of the equality (==) and inequality (!=) operators can have **enum** type; an **enum** value is converted in the same manner as an **int** value.

The operands of any relational operator can be two pointers to the same type. For the equality (==) and inequality (!=) operators, the result of the comparison reflects whether or not the two pointers address the same memory location. The result of pointer comparisons involving the other operators (<, >, <=, >=) reflects the relative position of two memory addresses.

Since the address of a given value is arbitrary, comparisons between the addresses of two unrelated values are generally meaningless. Comparisons between the addresses of different elements of the same array can be useful, however, since array elements are guaranteed to be stored in order from the first element to the last. The address of the first array element is "less than" the address of the last element.

A pointer value can be compared for equality (==) or inequality (!=) to the constant value 0. A pointer with a value of 0 does not point to a memory location: it is called a "null" pointer. A pointer value is equal to 0 only if it is explicitly given that value through assignment or initialization.

Examples

```
int x = 0, y = 0;

x < y           /* Example 1 */

x > y           /* Example 2 */

x <= y          /* Example 3 */

x >= y          /* Example 4 */

x == y          /* Example 5 */

x != y          /* Example 6 */
```

When x and y are equal, expressions 3, 4, and 5 have the value 1 and expressions 1, 2, and 6 have the value 0.

5.3.9 Bitwise Operators

The bitwise operators perform bitwise-AND (&), inclusive-OR (|), and exclusive-OR (^) operations. The operands of bitwise operators must have integral type, but their types can be different. The usual arithmetic conversions are performed; the type of the result is the type of the operands after conversion.

Bitwise AND (&)

The bitwise-AND (&) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1, the corresponding bit of the result is set to 1; otherwise, the corresponding result bit is set to 0.

Bitwise Inclusive OR (|)

The bitwise-inclusive-OR (|) operator compares each bit of its first operand to the corresponding bit of the second operand. If either of the compared bits is a 1, the corresponding bit of the result is set to 1. Otherwise, both bits are 0, and the corresponding result bit is set to 0.

Bitwise Exclusive OR (^)

The bitwise-exclusive-OR (^) operator compares each bit of its first operand to the corresponding bit of the second operand. If one of the compared bits is a 0 and the other bit is a 1, the corresponding bit of the result is set to 1; otherwise, the corresponding result bit is set to 0.

Examples

```
short i = 0xab00;
short j = 0xabcd;
short n;

n = i & j;              /* Example 1 */

n = i | j;              /* Example 2 */

n = i ^ j;              /* Example 3 */
```

The result assigned to `n` in the first example is the same as `i` (AB00 hexadecimal). The bitwise inclusive OR in the second example results in the value ABCD (hexadecimal), while the bitwise exclusive OR in the third example produces CD (hexadecimal).

5.3.10 Logical Operators

The logical operators perform logical-AND (**&&**) and logical-OR (**||**) operations. The operands of the logical operators must have integral, floating-point, or pointer type. The types of the operands can be different.

The operands of logical-AND and logical-OR expressions are evaluated left to right. If the value of the first operand is sufficient to determine the result of the operation, the second operand is not evaluated.

Logical operators do not perform the standard arithmetic conversions. Instead, they evaluate each operand in terms of its equivalence to zero. A pointer has a value of 0 only if it is explicitly set to 0 through assignment or initialization.

The result of a logical operation is either 0 or 1, as described next. The type of the result is **int**.

Logical AND (&&)

The logical-AND operator (**&&**) produces the value 1 if both operands have nonzero values. If either operand is equal to 0, the result is 0. If the first operand of a logical-AND operation has a value of 0, the second operand is not evaluated.

Logical OR (||)

The logical-OR operator (**||**) performs an inclusive OR on its operands. It produces the value 0 if both operands have 0 values; if either operand has a nonzero value, the result is 1. If the first operand of a logical-OR operation has a nonzero value, the second operand is not evaluated.

Examples

```
int x, y;

if (x < y && y < z)                                  /* Example 1 */
        printf ("x is less than z\n");

if (x == y || x == z)                                /* Example 2 */
        printf ("x is equal to either y or z\n");
```

In the first example, the `printf` function is called to print a message if `x` is less than `y` and `y` is less than `z`. If `x` is greater than `y`, the second operand (`y < z`) is not evaluated and nothing is printed. Note that this could cause problems in cases where the second operand contains side effects.

In the second example, a message is printed if `x` is equal to either `y` or `z`. If `x` is equal to `y`, the second operand (`x == z`) is not evaluated.

5.3.11 Sequential-Evaluation Operator

The sequential-evaluation operator (,) evaluates its two operands sequentially from left to right. The result of the operation has the value and type of the right operand. The types of the operands are unrestricted. No conversions are performed.

The sequential-evaluation operator (also called the "comma" operator) is typically used to evaluate two or more expressions in contexts that allow only one expression to appear.

Examples

```
/******************** Example 1 ********************/

for ( i = j = 1; i + j < 20; i += i, j--);

/******************** Example 2 ********************/

func_one(x, y + 2, z);
func_two((x--, y + 2), z);
```

In the first example, each operand of the **for** statement's third expression is evaluated independently. The left operand, `i += i,` is evaluated first, then `j--` is evaluated.

As shown in Example 2, the comma character is used in other contexts as a separator. In the function call to `func_one`, three arguments, separated by commas, are passed to the called function: `x`, `y + 2` and `z`. The use of the comma character as a separator must not be confused with its use as an operator; the two uses are completely different.

In the function call to `func_two`, parentheses force the compiler to interpret the first comma as the sequential-evaluation operator. This function call passes two arguments to `func_two`. The first argument is the result of the sequential-evaluation operation `(x--, y + 2)`, which has the value and type of the expression `y + 2`; the second argument is `z`.

5.3.12 Conditional Operator

C has one ternary operator, the conditional operator (? :). Its form is as follows:

operand1 ? *operand2* : *operand3*

The expression *operand1* is evaluated in terms of its equivalence to 0. It must have integral, floating-point, or pointer type. If *operand1* has a nonzero value, *operand2* is evaluated and the result of the expression is the value of *operand2*. If *operand1* evaluates to 0, *operand3* is evaluated, and the result of the expression is the value of *operand3*. Note that either *operand2* or *operand3* is evaluated, but not both.

The type of the result depends on the types of the second and third operands, as follows:

1. If the second and third operands have integral or floating-point type (their types can be different), the usual arithmetic conversions are performed. The type of the result is the type of the operands after conversion.
2. The second and third operands can have the same structure, union, or pointer type. The type of the result is the same structure, union, or pointer type.
3. One of the second or third operands can be a pointer and the other a constant expression with the value 0. The type of the result is the pointer type.

Example

```
j = (i < 0) ? (-i) : (i);
```

The above example assigns the absolute value of `i` to `j`. If `i` is less than 0, `-i` is assigned to `j`. If `i` is greater than or equal to 0, `i` is assigned to `j`.

5.4 Assignment Operators

C's assignment operators can both transform and assign values in a single operation. Using a compound-assignment operator to replace two separate operations can reduce code size and improve program efficiency. The assignment operators are as follows:

Operator	Operation Performed
++	Unary increment
--	Unary decrement
=	Simple assignment
*=	Multiplication assignment
/=	Division assignment
%=	Remainder assignment
+=	Addition assignment
-=	Subtraction assignment
<<=	Left-shift assignment
>>=	Right-shift assignment
&=	Bitwise-AND assignment
|=	Bitwise-inclusive-OR assignment
^=	Bitwise-exclusive-OR assignment

In assignment, the type of the right-hand value is converted to the type of the left-hand value. The specific path of the conversion depends on the two types and is outlined in detail in Section 5.7.

5.4.1 Lvalue Expressions

An assignment operation specifies that the value of the right-hand operand is to be assigned to the storage location named by the left-hand operand. Therefore, the left-hand operand of an assignment operation (or the single operand of a unary assignment expression) must be an expression referring to a memory location. Expressions that refer to memory locations are called "lvalue" expressions. A variable name is such an expression: the name of the variable denotes a storage location, while the value of the variable is the value residing at that location.

The following C expressions may be lvalue expressions:

- Identifiers of character, integer, floating-point, pointer, enumeration, structure, or union type
- Subscript ([ ]) expressions, except when a subscript expression evaluates to a pointer to an array or a pointer to a function
- Member-selection expressions (−> and .), if the selected member is one of the aforementioned expressions
- Unary-indirection (*) expressions, except when such expressions refer to arrays or functions
- Type casts to pointer types, as long as the size of the object does not change
- An lvalue expression in parentheses

5.4.2 Unary Increment and Decrement

The unary assignment operators (++ and −−) increment and decrement their operand, respectively. The operand must have integral, floating-point, or pointer type, and must be an lvalue expression.

Operands of integral or floating-point type are incremented or decremented by the integer value 1. The type of the result is the type of the operand. An operand of pointer type is incremented or decremented by the size of the object it addresses. An incremented pointer points to the next object; a decremented pointer points to the previous object.

An increment (++) or decrement (−−) operator can appear either before or after its operand. When the operator prefixes its operand, the operand is incremented or decremented and its new value is the result of the expression. When the operator postfixes its operand, the immediate result of the expression is the value of the operand *before* it is incremented or decremented. After that result is noted in context, the operand is incremented or decremented.

Examples

```
/******************** Example 1 ********************/

if (pos++ > 0)
        *ptt = *qtt;
```

```
/******************** Example 2 ********************/

if (line[--i] != '\n')
        return;
```

In the first example, the variable `pos` is compared to 0, then incremented.

In the second example, the variable `i` is decremented before it is used as a subscript to `line`.

5.4.3 Simple Assignment

The simple assignment operator (=) performs assignment. The right operand is assigned to the left operand; the conversion rules for assignment apply (see Section 5.7.1).

Example

```
double x;
int y;

x = y;
```

The value of `y` is converted to **double** type and assigned to `x`.

5.4.4 Compound Assignment

The compound assignment operators consist of the simple assignment operator combined with another binary operator. Compound assignment operators perform the operation specified by the additional operator, then assign the result to the left operand. A compound assignment expression such as

expression1 += *expression2*

can be understood as

expression1 = *expression1* + *expression2*

However, the compound assignment expression is not equivalent to the expanded version because the compound assignment expression evaluates *expression1* only once, while in the expanded version *expression1* is evaluated twice: in the addition operation and in the assignment operation.

Each compound assignment operator performs the conversions that the corresponding binary operator performs, and restricts the types of its operands accordingly. The result of a compound assignment operation has the value and type of the left operand.

Example

```
#define MASK    0xffff

n |= MASK;
```

In this example a bitwise-inclusive-OR operation is performed on `n` and `MASK`, and the result is assigned to `n`. The manifest constant `MASK` is defined with a #**define** preprocessor directive, discussed in Section 8.2.1.

5.5 Precedence and Order of Evaluation

The precedence and associativity of C operators affect the grouping and evaluation of operands in expressions. An operator's precedence is meaningful only in the presence of other operators having higher or lower precedence. Expressions with higher-precedence operators are evaluated first.

Table 5.1 summarizes the precedence and associativity of C operators, listing them in order of precedence from highest to lowest. Where several operators appear together in a line or large brace, they have equal precedence and are evaluated according to their associativity.

Table 5.1

Precedence and Associativity of C Operators

Symbol[a]	Type of Operation	Associativity
( ) [ ] . ->	Expression	Left to right
- ~ ! * & ++ -- **sizeof** *casts*	Unary[b]	Right to left
* / %	Multiplicative	Left to right
+ -	Additive	Left to right
<< >>	Shift	Left to right
< > <= >=	Relational (inequality)	Left to right
== !=	Relational (equality)	Left to right
&	Bitwise-AND	Left to right
^	Bitwise-exclusive-OR	Left to right
|	Bitwise-inclusive-OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
? :	Conditional	Right to left
= *= /= %= += -= <<= >>= &= |= ^=	Simple and compound assignment[c]	Right to left
,	Sequential evaluation	Left to right

[a] Operators are listed in descending order of precedence. Where several operators appear in the same line or in a large brace, they have equal precedence.

[b] All unary operators have equal precedence.

[c] All simple and compound assignment operators have equal precedence.

As Table 5.1 shows, operands consisting of a constant, an identifier, a string, a function call, a subscript expression, a member-selection expression, or a parenthetical expression have highest precedence and associate left to right. Type-cast conversions have the same precedence and associativity as the unary operators.

An expression can contain several operators with equal precedence. When several such operators appear at the same level in an expression, evaluation proceeds according to the associativity of the operator, either right to left or left to right. The result of expressions involving multiple occurrences of multiplication (*), addition (+), or binary bitwise (& | ^) operators at the same level is indifferent to the direction of evaluation. The compiler is free to evaluate such expressions in any order, even when parentheses in the expression appear to specify a particular order.

Important

Only the sequential-evaluation operator (,) and the logical-AND (&&) and logical-OR (||) operators guarantee a particular order of evaluation for the operands. The sequential-evaluation operator (,) is guaranteed to evaluate its operands from left to right. (Note that the comma separating arguments in a function call is not the same as the sequential-evaluation operator and does not provide any such guarantee.)

The logical operators also guarantee evaluation of their operands left to right. However, the logical operators evaluate the minimum number of operands necessary to determine the result of the expression. Thus, some operands of the expression may not be evaluated. For example, in the expression `x && y++`, the second operand, `y++`, is evaluated only if `x` is true (nonzero). Thus `y` is not incremented when `x` is false (0).

The examples below show the default groupings for several expressions:

Expression	Default Grouping
`a & b || c`	`(a & b) || c`
`a = b || c`	`a = (b || c)`
`q && r || s--`	`(q && r) || s--`

In the first example, the bitwise-AND operator (**&**) has higher precedence than the logical-OR operator (||), so `a & b` forms the first operand of the logical-OR operation.

In the second example, the logical-OR operator (||) has higher precedence than the simple assignment operator (=), so `b || c` is grouped as the right-hand operand in the assignment. Note that the value assigned to `a` is either 0 or 1.

The third example shows a correctly formed expression that may produce an unexpected result. The logical-AND operator (**&&**) has higher precedence than the logical-OR operator (||), so `q && r` is grouped as an operand. Since the logical operators guarantee evaluation of operands from left to right, `q && r` is evaluated before `s--`. However, if `q && r` evaluates to a nonzero value, `s--` is not evaluated, and `s` is not decremented. To correct this problem, `s--` should appear as the first operand of the expression or should be decremented in a separate operation.

The following example shows an illegal expression that produces a program error:

Illegal Expression	Default Grouping
`p == 0 ? p += 1: p += 2`	`(p == 0 ? p += 1 : p) += 2`

In this example, the equality operator (==) has the highest precedence, so `p == 0` is grouped as an operand. The ternary operator (**? :**) has the next-highest precedence. Its first operand is `p == 0` and its second operand is `p += 1`. However, the last operand of the ternary operator is considered to be `p` rather than `p += 2`, since this occurrence of `p` binds more closely to the ternary operator than it does to the compound assignment operator. A syntax error occurs because `+= 2` does not have a left-hand operand.

To prevent errors of this kind, and to produce more readable code, the use of parentheses is recommended. The preceding example can be corrected and clarified through the use of parentheses, as shown below:

```
(p == 0) ? (p += 1) : (p += 2)
```

5.6 Side Effects

"Side effects" are changes in the state of the machine that occur as a result of evaluating an expression. They occur whenever the value of a variable is changed. Any assignment operation has side effects, and any call to a function that contains assignment operations has side effects.

The order of evaluation of side effects is implementation dependent, except where the compiler guarantees a particular order of evaluation, as outlined in Section 5.5.

For example, side effects occur in the following function call:

```
add (i + 1, i = j + 2)
```

The arguments of a function call can be evaluated in any order. The expression `i + 1` may be evaluated before `i = j + 2`, or vice versa, with different results in each case.

Unary increment and decrement operations involve assignment and can cause side effects, as shown in the following example:

```
d = 0;
a = b++ = c++ = d++;
```

The value of `a` is unpredictable. The value of `d` (initially 0) could be assigned to `c`, then to `b`, and then to `a` before any of the variables are incremented. In this case `a` would be equal to 0.

A second method of evaluating this expression begins by evaluating the operand `c++ = d++`. The value of `d` (initially 0) is assigned to `c`, and then both `d` and `c` are incremented. Next, the value of `c`, which is now 1, is assigned to `b` and `b` is incremented. Finally, the incremented value of `b` is assigned to `a`; in this case, the final value of `a` is 2.

Since the C language does not define the order of evaluation of side effects, both of these evaluation methods are correct and either can be implemented. Statements that depend on a particular order of evaluation for side effects produce nonportable and unclear code.

5.7 Type Conversions

Type conversions occur when a value is assigned to a variable of a different type, when a value is explicitly cast to another type, when an operator converts the type of its operand or operands before performing an operation, and when a value is passed as an argument to a function. The rules governing each kind of conversion are outlined next.

5.7.1 Assignment Conversions

In assignment operations, the type of the value being assigned is converted to the type of the variable receiving the assignment. C allows conversions by assignment between integral and floating-point types, even when the conversion entails loss of information. The methods of carrying out the conversions depend upon the type, as follows.

5.7.1.1 Conversions from Signed Integral Types

A signed integer is converted to a shorter signed integer by truncating the high-order bits, and is converted to a longer signed integer by sign extension. Conversion of signed integers to floating-point values takes place without loss of information, except that some precision can be lost when a **long** value is converted to a **float**. In converting a signed integer to an unsigned integer, the signed integer is converted to the size of the unsigned integer and the result is interpreted as an unsigned value.

Conversions from signed integral types are summarized in Table 5.2. This table assumes that the **char** type is signed by default. If a compile-time option is used to change the default for the **char** type to unsigned, the conversions for the **unsigned char** type given in Table 5.3 apply.

Table 5.2

Conversions from Signed Integral Types

From	To	Method
char[a]	**short**	Sign extend
char	**long**	Sign extend
char	**unsigned char**	Preserve pattern; high-order bit loses function as sign bit
char	**unsigned short**	Sign extend to **short**; convert **short** to **unsigned short**
char	**unsigned long**	Sign extend to **long**; convert **long** to **unsigned long**
char	**float**	Sign extend to **long**; convert **long** to **float**
char	**double**	Sign extend to **long**; convert **long** to **double**
short	**char**	Preserve low-order byte
short	**long**	Sign extend
short	**unsigned char**	Preserve low-order byte
short	**unsigned short**	Preserve bit pattern; high-order bit loses function as sign bit
short	**unsigned long**	Sign extend to **long**; convert **long** to **unsigned long**
short	**float**	Sign extend to **long**; convert **long** to **float**
short	**double**	Sign extend to **long**; convert **long** to **double**
long	**char**	Preserve low-order byte
long	**short**	Preserve low-order word
long	**unsigned char**	Preserve low-order byte
long	**unsigned short**	Preserve low-order word
long	**unsigned long**	Preserve bit pattern; high-order bit loses function as sign bit
long	**float**	Represent as **float**; if **long** cannot be represented exactly, some loss of precision occurs
long	**double**	Represent as **double**; if **long** cannot be represented exactly as a **double**, some loss of precision occurs

[a] All **char** entries assume that the **char** type is signed by default.

Note

The **int** type is equivalent either to the **short** type or to the **long** type, depending on the implementation. Conversion of an **int** value proceeds as for a **short** or a **long**, whichever is appropriate.

5.7.1.2 Conversions from Unsigned Integral Types

An unsigned integer is converted to a shorter unsigned or signed integer by truncating the high-order bits. An unsigned integer is converted to a longer unsigned or signed integer by zero extending. Unsigned values are converted to floating-point values by converting first to a signed integer of the same size, then converting that signed value to a floating-point value.

When an unsigned integer is converted to a signed integer of the same size, no change in the bit pattern occurs. However, the value it represents changes if the sign bit is set.

Conversions from unsigned integral types are summarized in Table 5.3.

Table 5.3

Conversions from Unsigned Integral Types

From	To	Method
unsigned char	**char**	Preserve bit pattern; high-order bit becomes sign bit
unsigned char	**short**	Zero extend
unsigned char	**long**	Zero extend
unsigned char	**unsigned short**	Zero extend
unsigned char	**unsigned long**	Zero extend
unsigned char	**float**	Convert to **long**; convert **long** to **float**

Table 5.3 *(continued)*

From	To	Method
unsigned char	**double**	Convert to **long**; convert **long** to **double**
unsigned short	**char**	Preserve low-order byte
unsigned short	**short**	Preserve bit pattern; high-order bit becomes sign bit
unsigned short	**long**	Zero extend
unsigned short	**unsigned char**	Preserve low-order byte
unsigned short	**unsigned long**	Zero extend
unsigned short	**float**	Convert to **long**; convert **long** to **float**
unsigned short	**double**	Convert to **long**; convert **long** to **double**
unsigned long	**char**	Preserve low-order byte
unsigned long	**short**	Preserve low-order word
unsigned long	**long**	Preserve bit pattern; high-order bit becomes sign bit
unsigned long	**unsigned char**	Preserve low-order byte
unsigned long	**unsigned short**	Preserve low-order word
unsigned long	**float**	Convert to **long**; convert **long** to **float**
unsigned long	**double**	Convert to **long**; convert **long** to **double**

Note

The **unsigned int** type is equivalent either to the **unsigned short** type or to the **unsigned long** type, depending on the implementation. Conversion of an **unsigned int** value proceeds as for an **unsigned short** or an **unsigned long**, whichever is appropriate.

5.7.1.3 Conversions from Floating-Point Types

A **float** value converted to a **double** undergoes no change in value. A **double** converted to a **float** is represented exactly, if possible. If the value is too large to fit into a **float**, precision is lost.

A floating-point value is converted to an integer value by converting to a **long**. Conversions to other integer types occur as for a **long**. The decimal portion of the floating-point value is discarded in the conversion to a **long**; if the result is still too large to fit into a **long**, the result of the conversion is undefined.

Conversions from floating-point types are summarized in Table 5.4.

Table 5.4

Conversions from Floating-Point Types

From	To	Method
float	**char**	Convert to **long**; convert **long** to **char**
float	**short**	Convert to **long**; convert **long** to **short**
float	**long**	Truncate at decimal point; if result is too large to be represented as **long**, result is undefined
float	**unsigned short**	Convert to **long**; convert **long** to **unsigned short**
float	**unsigned long**	Convert to **long**; convert **long** to **unsigned long**
float	**double**	Change internal representation
double	**char**	Convert to **float**; convert **float** to **char**
double	**short**	Convert to **float**; convert **float** to **short**
double	**long**	Truncate at decimal point; if result is too large to be represented as **long**, result is undefined
double	**unsigned short**	Convert to **long**; convert **long** to **unsigned short**
double	**unsigned long**	Convert to **long**; convert **long** to **unsigned long**
double	**float**	Represent as a **float**. If **double** value cannot be represented exactly as **float**, loss of precision occurs; if value is too large to be represented as **float**, the result is undefined

5.7.1.4 Conversions to and from Pointer Types

A pointer to one type of value can be converted to a pointer to a different type. The result may be undefined, however, because of the alignment requirements and sizes of different types in storage.

In some implementations, the special keywords **near**, **far**, and **huge** are available to modify the size of pointers within a program. A pointer can be converted to a pointer of a different size; the path of the conversion is implementation dependent. For example, on an 8086 processor, the compiler might use a segment-register value to convert a 16-bit pointer to a 32-bit pointer. See your system documentation for information on pointer conversions.

A pointer value can be converted to an integral value. The path of the conversion depends on the size of the pointer and the size of the integral type, as follows:

- If the pointer is the same size as or larger than the integral type, the pointer behaves like an unsigned value in the conversion, except that it cannot be converted to a floating-point value.
- If the pointer is smaller than the integral type, the pointer is first converted to a pointer with the same size as the integral type, then converted to the integral type. The method of converting a pointer to a longer pointer is implementation dependent; see your system documentation for information on pointer conversions.

An integral type can be converted to a pointer type. If the integral type is the same size as the pointer type, the conversion simply causes the integral value to be treated as a pointer (an unsigned integer). If the size of the integral type is different from the size of the pointer type, the integral type is first converted to the size of the pointer, using the conversion paths given in Tables 5.2 and 5.3. It is then treated as a pointer value.

If the special keywords **near**, **far**, and **huge** are implemented, implicit conversions may be made on pointer values. In particular, the compiler may make assumptions about the default size of pointers and convert passed pointer values accordingly, unless a forward declaration is present to override the implicit conversion. See your system documentation for information on pointer conversions.

5.7.1.5 Conversions from Other Types

An **enum** value is an **int** value, by definition of the **enum** type. Conversions to and from an **enum** value proceed as for the **int** type. An **int** is equivalent to either a **short** or a **long**, depending on the implementation.

No conversions between structure or union types are allowed.

The **void** type has no value, by definition. Therefore, it cannot be converted to any other type, nor can any value be converted to **void** by assignment. A value can be explicitly cast to **void**, however, as discussed in Section 5.7.2.

5.7.2 Type-Cast Conversions

Explicit type conversions can be made by means of a type cast. A type cast has the form

(*type-name*)*operand*

where *type-name* specifies a particular type and *operand* is a value to be converted to the specified type. (Type names are discussed in Section 4.9.)

The conversion of *operand* occurs as though it had been assigned to a variable of the named type. The conversion rules for assignments (outlined in Section 5.7.1) apply to type casts as well. The type name **void** can be used in a cast operation, but the resulting expression cannot be assigned to any item.

5.7.3 Operator Conversions

The conversions performed by C operators depend on the operator and on the type of the operand or operands. Many operators perform the "usual arithmetic conversions," which are outlined in Section 5.3.1.

C permits some arithmetic with pointers. In pointer arithmetic, integer values are converted to express memory positions; see the discussions of additive operators (Section 5.3.6) and subscript expressions (Section 5.2.5) for information.

5.7.4 Function-Call Conversions

The type of conversion performed on the arguments in a function call depends on whether a forward declaration with declared argument types is present for the called function.

If a forward declaration is present, and it includes declared argument types, the compiler performs type checking. The type-checking process is outlined in detail in Section 7.4.1, "Actual Arguments."

If no forward declaration is present, or if the forward declaration omits the argument-type list, the only conversions performed on the arguments in the function call are the usual arithmetic conversions. These conversions are performed independently on each argument in the call. This means that a **float** value is converted to a **double**; a **char** or **short** value is converted to an **int**; and an **unsigned char** or **unsigned short** is converted to an **unsigned int**.

If the special keywords **near**, **far**, and **huge** are implemented, implicit conversions may also be made on pointer values passed to functions. These implicit conversions can be overridden by providing argument-type lists to allow the compiler to perform type checking. See your system documentation for information on pointer conversions.

Chapter 6
Statements

6.1 Introduction

The statements of a C program control the flow of program execution. In C, as in other programming languages, several kinds of statements are available to perform loops, to select other statements to be executed, and to transfer control. This chapter describes C statements in alphabetical order, as follows:

break statement

compound statement

continue statement

do statement

expression statement

for statement

goto statement

if statement

null statement

return statement

switch statement

while statement

C statements consist of keywords, expressions, and other statements. The following keywords appear in C statements:

break	**default**	**for**	**return**
case	**do**	**goto**	**switch**
continue	**else**	**if**	**while**

The expressions in C statements are the expressions discussed in Chapter 5, "Expressions and Assignments." Statements appearing within C statements may be any of the statements discussed in this chapter. A statement that forms a component of another statement is called the "body" of the enclosing statement. Frequently the statement body is a "compound" statement; that is, a single statement composed of one or more statements.

The compound statement is delimited by braces; all other C statements end with a semicolon.

Any C statement may be prefixed with an identifying label consisting of a name and a colon. Statement labels are recognized only by the **goto** statement and are therefore discussed in Section 6.8, "The goto and Labeled Statements."

When a C program is executed, its effect is the same as execution of the statements in order of their appearance in the program, except where a statement explicitly transfers control to another location.

6.2 The break Statement

Syntax

break;

Execution

The **break** statement terminates the execution of the smallest enclosing **do**, **for**, **switch**, or **while** statement in which it appears. Control passes to the statement following the terminated statement. A **break** statement appearing outside any **do**, **for**, **switch**, or **while** statement causes an error.

Within nested statements, the **break** statement terminates only the **do**, **for**, **switch**, or **while** statement immediately enclosing it. To transfer control out of the nested structure, a **return** or **goto** statement can be used.

Example

```
for (i = 0; i < LENGTH - 1; i++) {
        for (j = 0; j < WIDTH - 1; j++) {
                if (lines[i][j] == '\0') {
                        lengths[i] = j;
                        break;
                }
        }
}
```

The above example processes an array of variable-length strings stored in `lines`. The **break** statement causes an exit from the interior **for** loop after the terminating null character (**\0**) of each string is found and stored in `lengths [i]`. Control then returns to the outer **for** loop. The variable `i` is incremented and the process is repeated until `i` is greater than or equal to LENGTH.

6.3 The Compound Statement

Syntax

{
[[*declaration*]]
.
.
.
statement
[[*statement*]]
.
.
.
}

Execution

The effect of a compound statement's execution is that of the execution of its statements in order of their appearance, except where a statement explicitly transfers control to another location. The form and meaning of the declarations that can appear at the head of a compound statement are described in Chapter 4, "Declarations."

Example

```
if (i > 0) {
        line[i] = x;
        x++;
        i--;
}
```

A compound statement typically appears as the body of another statement, such as the **if** statement. In the above example, if `i` is greater than 0, all of the statements in the compound statement are executed in order.

Labeled Statements

Like other C statements, any of the statements in a compound statement can carry a label. Transfer into the compound statement by means of a **goto** is therefore possible. However, transferring into a compound statement is dangerous when the compound statement includes declarations that initialize variables. Declarations in a compound statement precede the executable statements, so transferring directly to an executable statement within the compound statement bypasses the initializations. The results are unpredictable.

6.4 The continue Statement

Syntax

continue;

Execution

The **continue** statement passes control to the next iteration of the **do**, **for**, or **while** statement in which it appears, bypassing any remaining statements in the **do**, **for**, or **while** statement body. Within a **do** or a **while** statement, the next iteration begins with the reevaluation of the **do** or **while** statement's expression. Within a **for** statement, the next iteration starts with the evaluation of the **for** statement's loop expression. It proceeds with the evaluation of the conditional expression and subsequent termination or reiteration of the statement body.

Example

```
while (i-- > 0) {
        x = f(i);
        if (x == 1)
                continue;
        y = x * x;
}
```

The statement body is executed if `i` is greater than 0. First `f(i)` is assigned to `x`; then, if `x` is equal to 1, the **continue** statement is executed. The rest of the statements in the body are ignored, and execution resumes at the top of the loop with the evaluation of `i-- > 0`.

6.5 The do Statement

Syntax

do
statement
while (*expression*);

Execution

The body of a **do** statement is executed one or more times until *expression* becomes false (0). First, the statement body is executed; then *expression* is evaluated. If *expression* is false, the **do** statement terminates and control passes to the next statement in the program. If *expression* is true (nonzero), the statement body is executed again, and *expression* is tested again. The statement body is executed repeatedly until *expression* becomes false.

The **do** statement may also terminate with the execution of a **break**, **goto**, or **return** statement within the statement body.

Example

```
do {
        y = f(x);
        x--;
} while (x > 0);
```

The two statements `y = f(x);` and `x--;` are executed, regardless of the initial value of `x`. Then `x > 0` is evaluated. If `x` is greater than 0, the statement body is executed again and `x > 0` is reevaluated. The statement body is executed repeatedly as long as `x` remains greater than 0. Execution of the **do** statement terminates when `x` becomes 0 or negative.

6.6 The Expression Statement

Syntax

expression;

Execution

The expression is evaluated, according to the rules outlined in Chapter 5, "Expressions and Assignments."

Examples

```
x = (y + 3);                /* Example 1 */

x++;                        /* Example 2 */

f(x);                       /* Example 3 */
```

In C, assignments are expressions; the value of the expression is the value being assigned (sometimes called the "right-hand value"). In the first example, `x` is assigned the value of `y + 3`. In the second example, `x` is incremented.

The third example shows a function-call expression. The value of the expression is the value, if any, returned by the function. If a function returns a value, the expression statement usually incorporates an assignment to store the returned value when the function is called. If the return value is not assigned, as in the example, the function call is executed but the return value, if any, is not used.

6.7 The for Statement

Syntax

for ([[*init-expression*]]; [[*cond-expression*]]; [[*loop-expression*]])
statement

Execution

The body of a **for** statement is executed zero or more times until the optional *cond-expression* becomes false. The *init-expression* and *loop-expression* are optional expressions that can be used to initialize and modify values during the **for** statement's execution.

The first step in the execution of the **for** statement is the evaluation of *init-expression*, if present. Next, *cond-expression* is evaluated, with three possible results:

1. If the conditional expression is true (nonzero), the statement body is executed; then *loop-expression*, if present, is evaluated. The process then begins again with the evaluation of *cond-expression*.
2. If the conditional expression is omitted, the conditional expression is considered true, and execution proceeds exactly as described above. A **for** statement lacking *cond-expression* terminates only upon the execution of a **break, goto,** or **return** statement within the statement body.
3. If the conditional expression is false, execution of the **for** statement terminates and control passes to the next statement in the program.

A **for** statement may also terminate with the execution of a **break, return,** or **goto** statement within the statement body.

Example

```
for (i = space = tab = 0; i < MAX; i++) {
        if (line[i] == '\x20')
                space++;
        if (line[i] == '\t') {
                tab++;
                line[i] ='\x20';
        }
}
```

The above example counts space (`'\x20'`) and tab (`'\t'`) characters in the array of characters named `line` and replaces each tab character with a space. First `i`, `space`, and `tab` are initialized to 0. Then `i` is compared to the constant `MAX`; if `i` is less than `MAX`, the statement body is executed. Depending on the value of `line[i]`, the body of one or neither of the **if** statements is executed. Then `i` is incremented and tested against `MAX`; the statement body is executed repeatedly as long as `i` is less than `MAX`.

6.8 The goto and Labeled Statements

Syntax

goto *name*;
.
.
.
name: *statement*

Execution

The **goto** statement transfers control directly to the statement specified by *name*. The labeled statement is executed immediately after the **goto** statement is executed. An error results if no statement with the given label resides in the same function or if an identical label appears before more than one statement in the same function.

A statement label is meaningful only to a **goto** statement; when a labeled statement is encountered in any other context, the statement is executed without regard to the label.

Example

```
if (errorcode > 0)
        goto exit;
.
.
.
exit:
        return (errorcode);
```

In the example, a **goto** statement transfers control to the point labeled `exit` when an error occurs.

Forming Labels

A label name is simply an identifier, formed by following the same rules that govern the construction of identifiers (see Section 2.4). Each statement label must be distinct from other statement labels and identifiers in the same function.

6.9 The if Statement

Syntax

if (*expression*)
    statement1
[**else**
    statement2]

Execution

The body of an **if** statement is executed selectively, depending on the value of *expression*. First, *expression* is evaluated. If *expression* is true (nonzero), the statement immediately following it is executed. If *expression* is false, the statement following the **else** keyword is executed. If *expression* is false and the **else** clause is omitted, the statement following *expression* is ignored. Control then passes from the **if** statement to the next statement in the program.

Example

```
if (i > 0)
        y = x/i;
else {
        x = i;
        y = f(x);
}
```

In the example, the statement `y = x/i;` is executed if `i` is greater than 0. If `i` is less than or equal to 0, `i` is assigned to `x` and `f(x)` is assigned to `y`. Note that the statement forming the **if** clause ends with a semicolon.

Nesting

C does not offer an "else if" statement, but the same effect is achieved by nesting **if** statements. An **if** statement may be nested in either the **if** clause or the **else** clause of another **if** statement.

When nesting **if** statements and **else** clauses, use braces to group the statements and clauses into compound statements that clarify your intent. In the absence of braces, the compiler resolves ambiguities by pairing each **else** with the most recent **if** lacking an **else**.

Examples

```
/******************** Example 1 ********************/

if (i > 0)              /* Without braces */
        if (j > i)
                x = j;
        else
                x = i;


/******************** Example 2 ********************/

if (i > 0) {            /* With braces */
        if (j > i)
                x = j;
}
else
        x = i;
```

In the first example, the **else** is associated with the inner **if** statement. If `i` is less than or equal to 0, no value is assigned to `x`.

In the second version, the braces surrounding the inner **if** statement make the **else** clause part of the outer **if** statement. If `i` is less than or equal to 0, `i` is assigned to `x`.

6.10 The Null Statement

Syntax

```
;
```

Execution

A null statement is a statement containing only a semicolon; it may appear wherever a statement is expected. Nothing happens when a null statement is executed.

Example

```
for (i = 0; i < 10; line[i++] = 0)
        ;
```

Statements such as **do, for, if,** and **while** require that an executable statement appear as the statement body. The null statement satisfies the syntax requirement in cases that do not need a substantive statement body. In the above example, the third expression of the **for** statement initializes the first 10 elements of `line` to 0. The statement body is a null statement, since no further statements are necessary.

Labeling a Null Statement

The null statement, like any other C statement, may be prefixed by an identifying label. To label an item that is not a statement, such as the closing brace of a compound statement, you can insert and label a null statement immediately preceding the item to get the same effect.

6.11 The return Statement

Syntax

return [[*expression*]];

Execution

The **return** statement terminates the execution of the function in which it appears and returns control to the calling function. Execution resumes in the calling function at the point immediately following the call. The value of *expression*, if present, is returned to the calling function. If *expression* is omitted, the return value of the function is undefined.

Example

```
main()
{
        void draw(int,int);
        long sq(int);
        .
        .
        .
        y = sq(x);
        draw(x, y);
        .
        .
        .
}

long sq(x)
int x;
{
        return (x * x);
}

void draw(x,y)
int x, y;
{
        .
        .
        .
        return;
}
```

The `main` function calls two functions, `sq` and `draw`. The `sq` function returns the value of `x * x` to `main`; the return value is assigned to `y`. The `draw` function is declared as a **void** function and does not return a value. An attempt to assign the return value of `draw` would cause an error.

By convention, parentheses enclose the *expression* of the **return** statement, as shown above. The language does not require the parentheses.

Omitting the Return Statement

If no **return** statement appears in a function definition, control automatically returns to the calling function after the last statement of the called function. The return value of the called function is undefined. If a return value is not required, the function should be declared to have **void** return type.

6.12 The switch Statement

Syntax

```
switch (expression) {
        [[declaration]]
        .
        .
        .
        [[case constant-expression :]]
        .
        .
        .
                [[statement]]
                .
                .
                .
        [[default :
                statement]]
        [[case constant-expression :]]
        .
        .
        .
                [[statement]]
                .
                .
                .
}
```

Execution

The **switch** statement transfers control to a statement within its body. The statement receiving control is the statement whose **case** *constant-expression* matches the value of the *expression* in parentheses. Execution of the statement body begins at the selected statement and proceeds through the end of the body or until a statement transfers control out of the body.

The **default** statement is executed if no **case** *constant-expression* is equal to the value of the **switch** *expression.* If the **default** statement is omitted, and no **case** match is found, none of the statements in the **switch** body is executed.

The **switch** expression is an integral value that must be the size of an **int** or shorter. It can also be an **enum** value. If the *expression* is shorter than an **int**, it is widened to an **int** value. Each **case** *constant-expression* is then cast to the type of the **switch** expression. The value of each **case** *constant-expression* must be unique within the statement body.

The **case** and **default** labels of the **switch** statement body are significant only in the initial test that determines the starting point for execution of the statement body. All statements appearing between the statement where execution starts and the end of the body are executed regardless of their labels, unless a statement transfers control out of the body entirely.

Declarations may appear at the head of the compound statement forming the **switch** body, but initializations included in the declarations are not performed. The effect of the **switch** statement is to transfer control directly to an executable statement within the body, bypassing the lines that contain initializations.

Examples

```
/******************** Example 1 ********************/

switch (c) {
           case 'A':
                   capa++;
           case 'a':
                   lettera++;
           default :
                   total++;
         }

/******************** Example 2 ********************/

switch (i) {
           case -1:
                  n++;
                  break;
           case 0 :
                  z++;
                  break;
           case 1 :
                  p++;
                  break;
         }
```

In the first example, all three statements of the **switch** body are executed if `c` is equal to `'A'`. Execution control is transferred to the first statement

(`capa++;`) and continues in order through the rest of the body. If `c` is equal to `'a'`, `lettera` and `total` are incremented. Only `total` is incremented if `c` is not equal to `'A'` or `'a'`.

In the second example, a **break** statement follows each statement of the **switch** body. The **break** statement forces an exit from the **switch** after one statement in the body is executed. If `i` is equal to −1, only `n` is incremented. The **break** following the statement `n++;` causes execution control to pass out of the **switch** body, bypassing the remaining statements. Similarly, if `i` is equal to 0, only `z` is incremented; if `i` is equal to 1, only `p` is incremented. The final **break** statement is not strictly necessary, since control will pass out of the body at the end of the compound statement, but it is included for consistency.

Multiple Labels

A statement may carry multiple **case** labels, as the following example shows:

```
case 'a' :
case 'b' :
case 'c' :
case 'd' :
case 'e' :
case 'f' :  hexcvt(c);
```

Although any statement within the body of the **switch** statement may be labeled, no statement is required to carry a label. Statements without labels may be intermingled freely with labeled statements. Keep in mind, however, that once the **switch** statement passes control to a statement within the body, all succeeding statements in the block are executed, regardless of their labels.

6.13 The while Statement

Syntax

while (*expression*)
    statement

Execution

The body of a **while** statement is executed zero or more times until *expression* becomes false (0). First, *expression* is evaluated. If the *expression* is initially false, the body of the **while** statement is never executed, and control passes from the **while** statement to the next statement in the program. If *expression* is true (nonzero), the body of the statement is executed. Following each execution of the statement body, *expression* is reevaluated; the body is executed repeatedly as long as *expression* remains true.

The **while** statement may also terminate with the execution of a **break**, **goto**, or **return** within the statement body.

Example

```
while (i >= 0) {
        string1[i] = string2[i];
        i--;
}
```

The above example copies characters from `string2` to `string1`. If `i` is greater than or equal to 0, `string2[i]` is assigned to `string1[i]` and `i` is decremented. When `i` reaches or falls below 0, execution of the **while** statement terminates.

Chapter 7
Functions

7.1 Introduction

A function is an independent collection of declarations and statements, usually designed to perform a specific task. C programs have at least one function, the main function, and they may have other functions. The sections of this chapter describe how to define, declare, and call C functions.

A function *definition* specifies the name of the function, its formal parameters, and the declarations and statements that define its action. The function definition can also give the return type of the function and its storage class.

A function *declaration* establishes the name, return type, and storage class of a function whose explicit definition is given at another point in the program. The number and types of arguments to the function can also be specified in the function declaration. This allows the compiler to compare the types of the actual arguments and the formal parameters of a function. Function declarations are optional for functions whose return type is **int**. To ensure correct behavior, functions with other return types must be declared before they are called.

A function *call* passes execution control from the calling function to the called function. The actual arguments, if any, are passed by value to the called function. Execution of a **return** statement in the called function returns control and possibly a value to the calling function.

7.2 Function Definitions

A function definition specifies the name, formal parameters, and body of a function. It may also define the function's return type and storage class. A function definition has the following form:

[*sc-specifier*][*type-specifier*] *declarator* ([*parameter-list*])
[*parameter-declarations*]
function-body

The *sc-specifier* gives the function's storage class, which must be either **static** or **extern**. The *type-specifier* and *declarator* together specify the

function's return type and name. The *parameter-list* is a list (possibly empty) of formal parameters to be used by the function. The *parameter-declarations* establish the types of the formal parameters. The *function-body* is a compound statement containing local variable declarations and statements. The following sections describe in detail the parts of the function definition.

7.2.1 Storage Class

The storage-class specifier in a function definition gives the function either **static** or **extern** storage class. A function with **static** storage class is visible only in the source file in which it is defined. All other functions, whether they are given **extern** storage class explicitly or implicitly, are visible throughout all the source files that constitute the program.

When the storage-class specifier is omitted from a function definition, the storage class defaults to **extern**. The **extern** storage-class specifier can be explicitly specified in the function definition, but it is not required.

The storage-class specifier is required in a function definition in only one case: when the function is declared elsewhere in the same source file with the **static** storage-class specifier.

The **static** storage-class specifier can also be used when defining a function previously declared in the same source file without a storage-class specifier. Normally, a function declared without a storage-class specifier defaults to the **extern** class. However, if the function definition explicitly specifies the **static** class, the function is given **static** class instead.

7.2.2 Return Type

The return type of a function defines the size and type of value returned by the function. The type declaration has the form

[[*type-specifier*]] *declarator*

where *type-specifier*, together with the *declarator*, defines the function's return type and name. If no *type-specifier* is given, the return type **int** is assumed.

The *type-specifier* can specify any fundamental, structure, or union type. The *declarator* consists of the function identifier, possibly modified to declare a pointer type. Functions cannot return arrays or functions, but they can return pointers to any type, including arrays and functions.

The return type given in the function definition must match the return type in declarations of the function elsewhere in the program. Functions with **int** return type do not have to be declared before they are called; functions with other return types cannot be called before they are either defined or declared.

A function's return value type is used only when the function returns a value, which occurs when a **return** statement containing an expression is executed. The expression is evaluated, converted to the return value type if necessary, and returned to the point of call. If no **return** statement is executed, or if the executed **return** statement does not contain an expression, the return value of the function is undefined. If the calling function expects a return value, the behavior of the program is also undefined.

Examples

```
/******************** Example 1 ********************/

        /* return type is int */
    static add (x, y)
    int x, y;
    {
        return (x+y);
    }

/********************* Example 2 *********************/

    typedef struct  {
        char name[20];
        int id;
        long class;
    } STUDENT;

          /* return type is STUDENT */
        STUDENT sortstu (a, b)
        STUDENT a, b;
    {
        return ( (a.id < b.id) ? a : b );
    }
```

```
/********************* Example 3 *********************/

      /* return type is char pointer */
    char *smallstr(s1, s2)
    char s1[], s2[];
    {
        int i;

        i=0;
        while ( s1[i] != '\0' && s2[i] != '\0' )
                        i++;
        if ( s1[i] == '\0' )
                return (s1);
        else
                return (s2);
    }
```

In the first example, the return type of `add` is **int** by default. The function has **static** storage class, which means it can be called only by functions in the same source file.

The second example defines the `STUDENT` type with a **typedef** declaration and defines the function `sortstu` to have `STUDENT` return type. The function selects and returns one of its two structure arguments.

The third example defines a function returning a pointer to an array of characters. The function takes two character arrays (strings) as arguments and returns a pointer to the shorter of the two strings. A pointer to an array points to the type of the array elements; thus, the return type of the function is a pointer to **char**.

7.2.3 Formal Parameters

Formal parameters are variables that receive values passed to a function by a function call. The formal parameters are declared in a parameter list at the beginning of the function declaration. The parameter list defines the names of the parameters and the order in which they take on values in the function call.

The parameter list consists of zero or more identifiers, separated by commas. The list must be enclosed in parentheses, even if no identifiers are given.

A comma followed by three periods (,...) can appear after the last identifier in the parameter list, indicating that the number of arguments to the function is variable. However, the function is expected to have at least as many parameters as there are identifiers before the last comma.

A parameter list can also consist of three periods (...) and no identifiers. This indicates that the number of parameters to the function is variable and may be zero.

Note

To maintain compatibility with previous versions, the compiler will also accept the comma character, without the trailing periods, at the end of the parameter list to indicate a variable number of arguments. A single comma can also be used instead of three periods to form the parameter list of a function taking zero or more arguments. Use of the comma is supported only for compatibility; using the three periods is recommended for new code.

Parameter declarations define the type and size of values stored in the formal parameters. These declarations have the same form as other variable declarations (see Section 4.4). A formal parameter can have any fundamental, structure, union, pointer, or array type.

A parameter can only have **auto** or **register** storage class. If no storage class is given, **auto** storage is assumed. If a formal parameter is named in the parameter list but is not declared, the parameter is assumed to have **int** type. Formal parameters can be declared in any order.

The identifiers of the formal parameters are used in the function body to refer to the values passed to the function. These identifiers cannot be used for variable declarations within the function body.

Only identifiers that appear in the parameter list can be declared as formal parameters. If the function has a variable number of arguments, the programmer is responsible for determining the number of arguments passed, and for retrieving additional arguments from the stack within the body of the function. See your system documentation for information on macros that can be used to do this in a portable way.

The type of each formal parameter should correspond to the type of the actual argument and to the type of the corresponding argument in the argument-type list for the function, if such a list is present. The compiler performs the usual arithmetic conversions independently on each formal parameter and on each actual argument, if necessary. After conversion, no formal parameter is shorter than an **int**, and no formal parameter has **float** type. This means, for example, that declaring a formal parameter as a **char** has the same effect as declaring it as an **int**.

If the **near**, **far**, and **huge** keywords are implemented, the compiler may also perform conversions on any pointer arguments to a function. The conversions performed depend on the default size of pointers in the program and the presence or absence of an argument-type list for the function. See your system documentation for specific information on pointer conversions.

The converted type of each formal parameter determines how the arguments placed on the stack by the function call are interpreted. A type mismatch between an actual and a formal parameter can cause the arguments on the stack to be misinterpreted. For example, if a 16-bit pointer is passed as an actual argument, then declared as a **long** formal parameter, the first 32 bits on the stack are interpreted as a **long** formal parameter. This error creates problems not only with the **long** formal parameter, but with any formal parameters that follow it. Errors of this kind can be detected through diligent use of argument-type lists in function declarations.

Example

```
struct student {
        char name[20];
        int id;
        long class;
        struct student *nextstu;
} student;

main()
{
        int match ( struct student *, char * );
        .
        .
        .
```

```
        if (match (student.nextstu, student.name) > 0) {
        .
        .
        .
        }
}

match ( r, n )
struct student *r;
char *n;
{
        int i = 0;

        while ( r->name[i] == n[i] )
                if ( r->name[i++] == '\0' )
                        return (r->id);
        return (0);
}
```

The example contains a structure type declaration, a forward declaration of the function `match`, a call to `match`, and the definition of the `match` function. Note that the same name, `student`, can be used without conflict both for the structure tag and for the structure variable name.

The `match` function is declared to have two arguments, the first a pointer to the `student` structure type and the second a pointer to a **char** type.

The two formal parameters of the `match` function are `r` and `n`. The parameter `r` is declared as a pointer to the `student` structure type; the parameter `n` is declared as a pointer to a **char** type.

The function is called with two arguments, both members of the `student` structure. Because there is a forward declaration of `match`, the compiler performs type checking between the actual arguments and the argument-type list and between the actual arguments and the formal parameters. Since the types match, no warnings or conversions are necessary.

Note that the array name given as the second argument in the call evaluates to a **char** pointer. The corresponding formal parameter is also declared as a **char** pointer, and is used in subscripted expressions as though it were an array identifier. Since an array identifier evaluates to a pointer expression, the effect of declaring the formal parameter as `char *n` is the same as declaring it `char n[ ]`.

Within the function, the local variable `i` is defined and used to monitor the current position in the array. The function returns the `id` structure member if the `name` member matches the array `n`; otherwise, it returns 0.

7.2.4 Function Body

The function body is simply a compound statement. The compound statement contains the statements that define the function's action and can also contain declarations of variables used by these statements. See Section 6.3 of Chapter 6, "Statements," for a discussion of compound statements.

All variables declared in the function body have **auto** storage type unless otherwise specified. When the function is called, storage space for the local variables is created and local initializations are performed. Execution control passes to the first statement in the compound statement and continues sequentially until a **return** statement or the end of the function body is encountered. Control then returns to the point of call.

A **return** statement containing an expression must be executed if the function is to return a value. The return value of a function is undefined if no **return** statement is executed or if the **return** statement does not include the optional expression.

7.3 Function Declarations

A function declaration defines the name, return type, and storage class of a given function, and may establish the type of some or all of the function's arguments. See Chapter 4, "Declarations," for a detailed description of the syntax of function declarations.

Functions can be declared implicitly or with forward declarations. The return type of a function declared either implicitly or with a forward declaration must agree with the return type specified in the function definition.

An implicit declaration occurs whenever a function is called without being previously defined or declared. The C compiler implicitly declares the called function to have **int** return type. By default, the function is declared to have **extern** storage class; the function definition can redefine the storage class to **static**, provided the function definition is given later in the same source file.

A forward declaration establishes the attributes of a function, allowing the declared function to be called before it is defined or allowing it to be called from another source file. If the storage-class specifier **static** is given in a forward declaration, the function has **static** class. The function definition must also specify the **static** class. If the storage-class specifier is **extern** or

is omitted, the function has **extern** class. However, the function definition can redefine the storage class as **static**, provided the function definition appears below the declaration in the same source file.

Forward declarations have several important uses. They establish the return type for functions that return any type of value but **int**. (Functions that return **int** values can also have forward declarations, but do not require them.) If a function with non-**int** return type is called before it is declared or defined, the results are unpredictable.

Forward declarations can be used to establish the types of arguments expected in a function call. The optional argument-type list of a forward declaration gives the type and number of arguments expected. (The number of arguments can vary.) The argument-type list is a list of type names corresponding to the expression list in the function call.

If no argument-type list is supplied, no type checking is performed. Type mismatches between actual arguments and formal parameters are silently accepted. Type checking is discussed further in Section 7.4.1, "Actual Arguments."

Forward declarations are also used to declare pointers to functions before the functions are defined.

Example

```
main()
{
        int a = 0, b = 1;
        float x = 2.0, y = 3.0;
        double realadd(double, double);

        a = intadd (a, b);
        x = realadd(x, y);
}

intadd(a, b)
int a, b;
{
        return (a + b);
}

double realadd(x, y)
double x, y;
{
        return (x + y);
}
```

In the example, the function `intadd` is implicitly declared to return an **int** value, since it is called before it is defined. The compiler does not check the types of the arguments in the call because no argument-type list is available.

The function `realadd` returns a **double** value instead of an **int**. The forward declaration of `realadd` in the `main` function allows the `realadd` function to be called before it is defined. Note that the definition of `realadd` matches the forward declaration by specifying the **double** return type.

The forward declaration of `realadd` also establishes the types of its two arguments; the actual arguments match the types given in the forward declaration and also match the types of the formal parameters.

7.4 Function Calls

A function call is an expression that passes control and zero or more actual arguments to a function. A function call has the form

expression([[*expression-list*]])

where *expression* evaluates to a function address and *expression-list* is a list of expressions (separated by commas) whose values, the actual arguments, are passed to the function. The *expression-list* can be empty.

When the function call is executed, the expressions in the function expression list are copied, converted as necessary, and then passed to formal parameters of the called function. The first expression in the list always corresponds to the first formal parameter of the function, the second expression corresponds to the second formal parameter, and so on through the end of the list. Since the called function works with copies of the actual arguments, any changes it makes to the arguments are not reflected in the original values from which the copies were made.

Execution control then passes to the first statement in the function. The execution of a **return** statement in the body of the function returns control and possibly a value to the calling function. If no **return** statement is executed, control returns to the caller after the last statement of the called function is executed, and the return value is undefined.

Important

The expressions in the function call's expression list can be evaluated in any order, so expressions with side effects have unpredictable results. The only guarantee the compiler makes is that all side effects in the expression list are evaluated before control passes to the called function.

The only requirement in calling a function is for the expression before the parentheses to evaluate to a function address. This means that a function can be called through any function-pointer expression. It may be helpful to remember that a function is called in the same manner in which it is declared. For instance, when declaring a function, the name of the function is given, followed by an argument-type list in parentheses. To call the function, only the name of the function is required, followed by an expression list in parentheses. The indirection operator (*) is not required to call the function because the name of the function evaluates to the function address.

The same principle applies when calling a function through a pointer. For example, suppose a function pointer is declared as follows:

```
int (*fpointer)(int, int);
```

The identifier `fpointer` is declared to point to a function taking two **int** arguments and returning an **int** value. A function call through `fpointer` might look like this:

```
(*fpointer)(3,4)
```

The indirection operator (*) is used to obtain the address of the function to which `fpointer` points. The function address is then used to call the function.

Examples

```
/********************* Example 1 *********************/

    double *realcomp(double, double);
    double a, b, *rp;
    .
    .
    .
    rp = realcomp(a, b);

/********************* Example 2 *********************/

    main ()
    {
        long lift(int), step(int), drop(int);
        void work (int, long (*)(int));
        int select, count;
        .
        .
        .
        select = 1;
        switch ( select ) {
                case 1: work(count, lift);
                        break;

                case 2: work(count, step);
                        break;

                case 3: work(count, drop);

                default:
                        break;
        }
    }

    void work ( n, func )
    int n;
    long (*func)(int);
    {
        int i;
        long j;

        for (i = j = 0; i < n; i++)
                j += (*func)(i);
    }
```

In the first example, the `realcomp` function is called in the statement `rp = realcomp(a, b);`. Two **double** arguments are passed to the `realcomp` function; the return value, a pointer to a **double**, is assigned to `rp`.

In the second example, the function call

```
work (count, lift);
```

in `main` passes an integer variable and the address of the function `lift` to the function `work`. Note that the function address is passed simply by giving the function identifier, since a function identifier evaluates to a pointer expression. To use a function identifier in this way, the function must be declared or defined before the identifier is used; otherwise, the identifier is not recognized. In this case, a forward declaration for `work` is given at the beginning of the `main` function.

The formal parameter `func` in `work` is declared to be a pointer to a function taking one **int** argument and returning a **long**. The parentheses around the parameter name are required; without them, the declaration would specify a function returning a pointer to a **long**.

The function `work` calls the selected function by using the following function call:

```
(*func)(i);
```

One argument, `i`, is passed to the called function.

7.4.1 Actual Arguments

An actual argument can be any value with fundamental, structure, union, or pointer type. Although arrays and functions cannot be passed as parameters, pointers to these items can be passed.

All actual arguments are passed by value. A copy of the actual argument is assigned to the corresponding formal parameter. The function uses this copy without affecting the variable from which it was originally derived.

Pointers provide a way to access a value by reference from a function. Since a pointer to a variable holds the address of the variable, the function can use this address to access the value of the variable. Pointer arguments allow a function to access arrays and functions, even though arrays and functions cannot be passed as arguments.

The expressions in a function call are evaluated and converted as follows:

- If an argument-type list is present, then for each actual argument in the function call, the usual arithmetic conversions are performed independently on the corresponding type in the argument-type list, and the actual argument is converted to that type. Next, the converted expression is compared with the type of the formal parameter that has the same place in the parameter list that the expression has in the expression list. (The formal parameters also undergo the usual arithmetic conversions before the comparison.) No conversions are performed, but the compiler produces warning messages as if the expressions were assigned to the formal parameters.
- If no argument-type list is present, or if there are more actual arguments than there are type names in the argument-type list, the usual arithmetic conversions are performed independently on each actual argument that lacks a corresponding type name.

If the **near**, **far**, and **huge** keywords are implemented, implementation-dependent conversions on pointer arguments may also be performed. See your system documentation for information on pointer conversions.

The number of expressions given in the expression list must match the number of formal parameters, unless the function's forward declaration and possibly its definition explicitly specify a variable number of arguments. In this case, the compiler checks as many arguments as there are type names in the argument-type list and converts them, if necessary, as described above.

If the argument-type list contains the special type name **void**, the compiler expects zero actual arguments in the function call and zero formal parameters. It produces a warning message if it finds otherwise.

The type of each formal parameter also undergoes the usual arithmetic conversions. The converted type of each formal parameter determines how the arguments on the stack are interpreted; if the type of the formal parameter does not match the type of the actual argument, the data on the stack can be misinterpreted.

Note

Type mismatches between actual and formal parameters can produce serious errors, particularly when the mismatches entail size differences. Keep in mind that these errors are not detected unless an argument-type list is given in the forward declaration of the function.

Example

```
main ()
{
        void swap (int *, int *);
        int x, y;
        .
        .
        .
        swap(&x, &y);
}

void swap (a, b)
int *a, *b;
{
        int t;

        t = *a;
        *a = *b;
        *b = t;
}
```

In the above example, the `swap` function is declared in `main` to have two arguments, both pointers to integers. The formal parameters `a` and `b` are also declared as pointers to integer variables. In the function call

```
swap (&x, &y)
```

the address of `x` is stored in `a` and the address of `y` is stored in `b`. Now two names, or "aliases," exist for the same location. References to `*a` and `*b` in `swap` are effectively references to `x` and `y` in `main`. The assignments within `swap` change the contents of `x` and `y`.

The compiler performs type checking on the arguments to `swap` because an argument-type list is present in the forward declaration of `swap`. The types of the actual arguments match both the argument-type list and the formal parameters.

7.4.2 Calls with a Variable Number of Arguments

To call a function with a variable number of arguments, the programmer simply gives any number of arguments in the function call. In the forward declaration of the function (if there is one), a variable number of arguments is specified by placing a comma followed by three periods (,...) at the end of the argument-type list (see Section 4.5 of Chapter 4, "Declarations"). One argument must be present in the function call for each type name specified in the argument-type list. If only the three periods (but no type names) are given, no arguments are required when calling the function.

Similarly, the parameter list in the function definition can end with a comma followed by three periods (,...) to indicate a variable number of arguments. If the parameter list contains only three periods (...), the number of parameters is variable and may be zero. See Section 7.2, "Function Definitions," for more information on the form of the parameter list.

Note

To maintain compatibility with previous versions, the compiler will also accept the comma character, without the trailing periods, at the end of the argument-type list or parameter list to indicate a variable number of arguments. A single comma can also be used instead of three periods to form the argument-type list or parameter list of a function taking zero or more arguments. Use of the comma is supported only for compatibility; use of the three periods is recommended for new code.

All the arguments given in the function call are placed on the stack. The number of formal parameters declared for the function determines how many of the arguments are taken from the stack and assigned to the formal parameters. The programmer is responsible for retrieving any additional arguments from the stack and for determining how many arguments are present. See your system documentation for information about macros that can be used to handle a variable number of arguments in a portable way.

7.4.3 Recursive Calls

Any function in a C program can be called recursively. A function can therefore call itself. The C compiler allows any number of recursive calls to a function. On each call, new storage is allocated for the formal parameters and for the **auto** and **register** variables so that their values in previous, unfinished calls are not overwritten. Previous parameters are inaccessible to all versions of the function except the version in which they were created.

Note that variables declared with global storage do not require new storage with each recursive call. Their storage exists for the lifetime of the program. Each reference to such a variable accesses the same storage area.

Although the C compiler defines no limit for the number of times a function can be called recursively, the operating environment may impose a practical limit. Since each recursive call requires additional stack memory, too many recursive calls can cause a stack overflow.

Chapter 8
Preprocessor Directives and Pragmas

8.1 Introduction

A "preprocessor directive" is an instruction intended for the C preprocessor. The C preprocessor is a text processor used to manipulate the text of a source file as the first phase of compilation. The compiler ordinarily invokes the preprocessor in its first pass, but the preprocessor can also be invoked separately to process text without compiling.

Preprocessor directives are typically used to make source programs easy to modify and to compile in different execution environments. Directives in the source file instruct the preprocessor to perform specific actions. For example, the preprocessor can replace tokens in the text, insert the contents of other files into the source file, and suppress compilation of a portion of the file by removing sections of text.

The C preprocessor recognizes the following directives:

#define	**#if**	**#line**
#elif	**#ifdef**	**#undef**
#else	**#ifndef**	
#endif	**#include**	

The number sign (#) must be the first non-white-space character on the line containing the directive; white-space characters can appear between the number sign and the first letter of the directive. Some directives are followed by arguments or values, as described below. Directives can appear anywhere in a source file, but they apply only to the remainder of the source file in which they appear.

A "pragma" is a "pragmatic," or practical, instruction to the C compiler. Pragmas are embedded in C source files and are typically used to control the actions of the compiler in a particular portion of a program without affecting the program as a whole. Section 8.6 describes the syntax for pragmas. However, the particular pragmas that are available, and their meanings, are defined by the implementation. See your system documentation for information on the use and effects of pragmas.

8.2 Manifest Constants and Macros

The # **define** directive is typically used to associate meaningful identifiers with constants, keywords, and commonly used statements or expressions. Identifiers that represent constants are called "manifest constants." Identifiers that represent statements or expressions are called "macros."

Once an identifier is defined, it cannot be redefined to a different value without first removing the definition. However, the identifier can be redefined with exactly the same definition. Thus, a program is allowed to contain more than one occurrence of the same definition.

The # **undef** directive removes the definition of an identifier. Once the definition has been removed, the identifier can be redefined to a different value. Sections 8.2.1 and 8.2.2 discuss the # **define** and # **undef** directives, respectively.

Macros can be defined to look and act like function calls. Because macros do not generate actual function calls, replacing function calls with macros can improve execution time. However, macros create problems if they are not defined and used with care. Macro definitions with arguments may require the use of parentheses to preserve the proper precedence in an expression. In addition, macros may not correctly handle expressions with side effects. See the examples in Section 8.2.1 for more information.

8.2.1 The # define Directive

Syntax

define *identifier text*
define *identifier*(*parameter-list*) *text*

The # **define** directive substitutes the given *text* for subsequent occurrences of the specified *identifier* in the source file. The *identifier* is replaced only when it forms a token. (Tokens are described in Chapter 2, "Elements of C," and in Appendix B, "Syntax Summary.") For instance, the *identifier* is not replaced when it occurs within a string or as part of a longer identifier.

If a *parameter-list* appears after the *identifier*, the # **define** directive replaces each occurrence of *identifier*(*parameter-list*) with a version of *text* modified by substituting actual arguments for formal parameters.

The *text* consists of a series of tokens, such as keywords, constants, or complete statements. One or more white-space characters must separate the *text* from the *identifier* (or from the closing parenthesis of the *parameter-list*). If the text is longer than one line, it can be continued onto the next line by preceding the new-line character with a backslash (\).

The *text* can also be empty. The effect of this option is to remove instances of the given *identifier* from the source file. The *identifier* is still considered defined, however, and yields the value 1 when tested with the #**if** directive (discussed in Section 8.4.1).

The *parameter-list*, when given, consists of one or more formal parameter names separated by commas. Each name in the list must be unique, and the list must be enclosed in parentheses. No spaces between the *identifier* and the opening parenthesis are allowed.

Formal parameter names appear in *text* to mark the places where actual values will be substituted. Each parameter name can occur more than once in the *text*, and the names can appear in any order.

The actual arguments following an instance of the *identifier* in the source file are matched to the formal parameters of the *parameter-list*, and the *text* is modified by replacing each formal parameter with the corresponding actual argument. The actual argument list and the formal *parameter-list* must have the same number of arguments.

Arguments with side effects sometimes cause macros to produce unexpected results. A macro definition may contain more than one occurrence of a given formal parameter, and if that formal parameter is replaced by an expression with side effects, the expression, with its side effects, is evaluated more than once (see Example 4 below).

Examples

```
/******************** Example 1 ********************/

#define WIDTH       80
#define LENGTH      (WIDTH + 10)

/******************** Example 2 ********************/

#define FILEMESSAGE "Attempt to create file \
failed because of insufficient space"
```

```
/******************** Example 3 ********************/

#define REG1        register
#define REG2        register
#define REG3

/******************** Example 4 ********************/

#define MAX(x,y)    ((x) > (y)) ? (x) : (y)

/******************** Example 5 ********************/

#define MULT(a,b)   ((a) * (b))
```

Example	Description

1	The first example defines the identifier `WIDTH` as the integer constant 80, and defines `LENGTH` in terms of `WIDTH` and the integer constant 10. Each occurrence of `LENGTH` is replaced with `( WIDTH + 10)`, which is in turn replaced with the expression `(80 + 10)`. The parentheses around `WIDTH + 10` are important because they control the interpretation in a statement such as the following:
	`var = LENGTH * 20;`
	After the preprocessing stage the statement becomes
	`var = (80 + 10) * 20;`
	or 1800. Without parentheses, the result is
	`var = 80 + 10 * 20;`
	which evaluates to 280 because the multiplication operator (*) has higher precedence than the addition operator (+).

2	The second example defines the identifier `FILEMESSAGE`. The definition is extended to a second line by using the backslash escape character (\).
3	The third example defines three identifiers, `REG1`, `REG2`, and `REG3`. `REG1` and `REG2` are defined as the keyword **register**. The definition of `REG3` is empty, so each occurrence of `REG3` is removed from the source file. These directives can be used to ensure that the program's most important

variables (declared with REG1 and REG2) are given **register** storage. See the discussion of the #**if** directive in Section 8.4.1 for an expanded version of this example.

4 The fourth example defines a macro named MAX. Each occurrence of the identifier MAX following the definition in the source file is replaced by the expression ((x) > (y)) ? (x) : (y), where actual values replace the parameters x and y. For example, the occurrence

```
MAX(1,2)
```

is replaced with

```
((1) > (2)) ? (1) : (2)
```

and the occurrence

```
MAX(i,s[i])
```

is replaced with

```
((i) > (s[i])) ? (i) : (s[i])
```

This macro is easier to read than the corresponding expression, making the source program easier to understand.

Note that arguments with side effects may cause this macro to produce unexpected results. For example, the occurrence MAX(i, s[i++]) is replaced with ((i) > (s[i++])) ? (i) : (s[i++]). The expression (s[i++]) is evaluated twice, so by the time the ternary expression has been fully evaluated, i has increased by 2. The result of the ternary expression is unpredictable, since the operands of the ternary expression can be evaluated in any order, and the value of i varies depending on the evaluation order.

5 The fifth example defines the macro MULT. Once the macro is defined, an occurrence such as MULT(3, 5) is replaced by (3) * (5). The parentheses around the parameters are important because they control the interpretation when complex expressions form the arguments to the macro.

For instance, the occurrence `MULT(3 + 4, 5 + 6)` is replaced by `(3 + 4) * (5 + 6)`, which evaluates to 77. Without the parentheses, the result is `3 + 4 * 5 + 6`, which evaluates to 29 because the multiplication operator (*) has higher precedence than the addition operator (+).

8.2.2 The #undef Directive

Syntax

undef *identifier*

The #**undef** directive removes the current definition of *identifier*. The preprocessor ignores subsequent occurrences of *identifier*. To remove a macro definition using #**undef**, give only the macro *identifier*; do not give a parameter list.

The #**undef** directive can also be applied to an identifier that has no previous definition. This ensures that the identifier is undefined.

The #**undef** directive is typically paired with a #**define** directive to create a region in a source program in which an identifier has a special meaning. For example, a specific function of the source program can use manifest constants to define environment-specific values that do not affect the rest of the program. The #**undef** directive also works with the #**if** directive (see Section 8.4.1) to control compilation of portions of the source program.

Example

```
#define WIDTH           80
#define ADD(X,Y)        (X) + (Y)
.
.
.
#undef WIDTH
#undef ADD
```

In this example, the #**undef** directive removes definitions of a manifest constant and a macro. Note that only the identifier of the macro is given.

8.3 Include Files

Syntax

#include "*pathname*"
#include <*pathname*>

The **#include** directive adds the contents of a given "include file" to another file. Constant and macro definitions can be organized into include files and added to any source file by using **#include** directives. Include files are also useful for incorporating declarations of external variables and complex data types. The types need only be defined and named once in an include file created for that purpose.

The **#include** directive tells the preprocessor to treat the contents of the named file as if they appeared in the source program at the point of the directive. The new text can also contain preprocessor directives. The preprocessor carries out directives in the new text, then continues processing the original text of the source file.

The *pathname* is a file name optionally preceded by a directory specification. It must name an existing file. The syntax of the file specification depends on the specific operating system on which the program is compiled.

The preprocessor uses the concept of a "standard" directory or directories to search for included files. The location of the standard directories for include files depends on the implementation and the operating system. See your system documentation for a definition of the standard directories.

The preprocessor stops searching as soon as it finds a file with the given name. If a complete, unambiguous path name for the include file is given, either in double quotation marks (" ") or in angle brackets (< >), the preprocessor searches only that path name and ignores the standard directories.

If the file specification does not give a complete path name, and the file specification is enclosed in double quotation marks, the preprocessor first searches for the file in the same directory as the including file (the "current working directory"). The preprocessor then searches directories specified in the compiler command line and finally searches the standard directories.

If the file specification is enclosed in angle brackets, the preprocessor does not search the current working directory. It begins by searching for the file in directories specified in the compiler command line and then searches the standard directories.

An #**include** directive can be nested; in other words, the directive can appear in a file named by another #**include** directive. When the preprocessor encounters the nested #**include** directive, it processes the named file and inserts it into the current file. The preprocessor uses the same search procedures outlined above in searching for nested include files.

The new file can also contain #**include** directives. Nesting can continue up to 10 levels. Once the nested #**include** is processed, the preprocessor continues to insert the enclosing include file into the original source file.

Examples

```
#include <stdio.h>              /* Example 1 */

#include "defs.h"               /* Example 2 */
```

The first example adds the contents of the file named `stdio.h` to the source program. The angle brackets cause the preprocessor to search the standard directories for `stdio.h`, after searching directories specified in the command line.

The second example adds the contents of the file specified by `defs.h` to the source program. The double quotation marks mean that the directory containing the current source file is searched first.

8.4 Conditional Compilation

This section describes the syntax and use of directives that control "conditional compilation." These directives allow for suppressing compilation of portions of a source file by testing a constant expression or identifier to determine which text blocks are passed on to the compiler and which are removed from the source file in the preprocessing stage.

8.4.1 The #if, #elif, #else, and #endif Directives

Syntax

```
# if restricted-constant-expression
    [[ text ]]
[[ # elif restricted-constant-expression
    text ]]
[[ # elif restricted-constant-expression
    text ]]
.
.
.
[[ # else
    text ]]
# endif
```

The #**if** directive, together with the #**elif**, #**else**, and #**endif** directives, controls compilation of portions of a source file. Each #**if** directive in a source file must be matched by a closing #**endif** directive. Zero or more #**elif** directives can appear between the #**if** and #**endif** directives, but at most one #**else** directive is allowed. The #**else** directive, if present, must be the last directive before #**endif.**

The preprocessor selects one of the given blocks of *text* for further processing. A *text* block is any sequence of text. It can occupy more than one line. Usually the *text* block is program text that has meaning to the compiler or the preprocessor. However, this is not a requirement; the preprocessor can be used to process any kind of text.

The selected *text* is processed by the preprocessor and passed to the compiler. If the *text* contains preprocessor directives, those directives are carried out.

Any text blocks not selected by the preprocessor are removed from the file in the preprocessing stage and are therefore not compiled.

The preprocessor selects a single *text* block by evaluating the *restricted-constant-expressions* following each #**if** or #**elif** directive until a true (nonzero) *restricted-constant-expression* is found. All *text* between the first true *restricted-constant-expression* and the next number sign (#) is selected.

If no *restricted-constant-expression* is true, or if there are no #**elif** directives, the preprocessor selects the text after the #**else** clause. If the #**else** clause is omitted, and no *restricted-constant-expression* in the #**if** block is true, no text is selected.

Each *restricted-constant-expression* follows the rules for restricted constant expressions discussed in Section 5.2.10 of Chapter 5, "Expressions and Assignments." Such expressions cannot contain **sizeof** expressions, type casts, or enumeration constants, but they can contain the special constant expression **defined(***identifier***)**. This constant expression is considered true (nonzero) if the given *identifier* is currently defined; otherwise, the condition is false (0). An identifier defined as empty text is considered defined.

The #**if**, #**elif**, #**else**, and #**endif** directives can nest in the text portions of other #**if** directives. When nested, each #**else**, #**elif**, and #**endif** directive belongs to the closest preceding #**if** directive.

Examples

```
/******************** Example 1 ********************/

#if defined(CREDIT)
    credit();
#elif defined(DEBIT)
    debit();
#else
    printerror();
#endif

/******************** Example 2 ********************/

#if DLEVEL > 5
    #define SIGNAL  1
    #if STACKUSE == 1
                    #define STACK   200
    #else
                    #define STACK   100
    #endif
#else
    #define SIGNAL  0
    #if STACKUSE == 1
                    #define STACK   100
    #else
                    #define STACK   50
    #endif
#endif
```

```
/******************** Example 3 ********************/

#if DLEVEL == 0
    #define STACK 0
#elif DLEVEL == 1
    #define STACK 100
#elif DLEVEL > 5
    display( debugptr );
#else
    #define STACK 200
#endif

/******************** Example 4 ********************/

#define REG1    register
#define REG2    register

#if defined(M_86)
    #define REG3
    #define REG4
    #define REG5
#else
    #define REG3    register
    #if defined(M_68000)
            #define REG4    register
            #define REG5    register
    #endif
#endif
```

In the first example, the #**if** and #**endif** directives control compilation of one of three function calls. The function call to `credit` is compiled if the identifier `CREDIT` is defined. If the identifier `DEBIT` is defined, the function call to `debit` is compiled. If neither identifier is defined, the call to `printerror` is compiled. Note that `CREDIT` and `credit` are distinct identifiers in C because their cases are different.

The next two examples assume a previously defined manifest constant, `DLEVEL`. The second example shows two sets of nested #**if**, #**else**, and #**endif** directives. The first set of directives is processed only if `DLEVEL > 5` is true. Otherwise, the second set is processed.

In the third example, #**elif** and #**else** directives are used to make one of four choices, based on the value of `DLEVEL`. The manifest constant `STACK` is set to 0, 100, or 200, depending on the definition of `DLEVEL`. If `DLEVEL` is greater than 5, `display(debugptr);` is compiled and `STACK` is not defined.

The fourth example uses preprocessor directives to control the meaning of **register** declarations in a portable source file. The compiler assigns register storage to variables in the same order in which the **register** declarations appear in the source file. If a program contains more **register** declarations than the machine can accommodate, the compiler honors earlier declarations over later ones. Loss of efficiency can occur if the variables declared later are more heavily used.

The definitions listed above can be used to give priority to the most important register declarations. REG1 and REG2 are defined as the **register** keyword to declare **register** storage for the two most important variables in the program. For example, in the following fragment, b and c have higher priority than a or d.

```
func(a)

REG3 int a;

{
        REG1 int b;
        REG2 int c;
        REG4 int d;
        .
        .
        .
}
```

When M_86 is defined, the preprocessor removes the REG3 identifier from the file by replacing it with empty text; this prevents a from receiving **register** storage at the expense of b and c. When M_68000 is defined, all four variables are declared to have **register** storage. When neither M_86 nor M_68000 is defined, a, b, and c are declared with **register** storage.

8.4.2 The #ifdef and #ifndef Directives

Syntax

 # ifdef *identifier*
 # ifndef *identifier*

The **#ifdef** and **#ifndef** directives accomplish the same task as the **#if** directive used with **defined(***identifier***)**. These directives can be used anywhere **#if** can be used, and are provided only for compatibility with previous versions of the language. The **defined(***identifier***)** constant expression used with the **#if** directive is preferred.

When the preprocessor encounters an #**ifdef** directive, it checks to see whether the *identifier* is currently defined. If so, the condition is true (nonzero); otherwise, the condition is false (0).

The #**ifndef** directive checks for the opposite condition checked by #**ifdef**. If the identifier has not been defined (or its definition has been removed with #**undef**), the condition is true (nonzero). Otherwise, the condition is false (0).

8.5 Line Control

Syntax

line *constant* [["*filename*"]]

The #**line** directive instructs the preprocessor to change the compiler's internally stored line number and file name to a given line number and file name. The compiler uses the internally stored line number and file name to refer to errors encountered during compilation. The line number normally refers to the current input line; the file name refers to the current input file. The line number is increased after each line is processed.

Changing the line number and file name causes the compiler to ignore the previous values and to continue processing with the new values. The #**line** directive is typically used by program generators to cause error messages to refer to the original source file instead of the generated program.

The *constant* value in the #**line** directive is any integer constant. The *filename* can be any combination of characters and must be enclosed in double quotation marks (" "). If *filename* is omitted, the previous file name remains unchanged.

The current line number and file name are always available through the predefined identifiers __**LINE**__ and __**FILE**__. The __**LINE**__ and __**FILE**__ identifiers can be used to insert self-descriptive error messages into the program text.

The __**FILE**__ identifier contains a string representing the file name, surrounded by double quotation marks (""). Thus, it is not necessary to enclose the __**FILE**__ identifier in quotation marks when using it as a string.

Examples

```
/******************** Example 1 ********************/

#line 151 "copy.c"

/******************** Example 2 ********************/

#define ASSERT(cond)        if(!cond)\
{printf("assertion error line %d, file(%s)\n", \
__LINE__, __FILE__ );} else ;
```

In the first example, the internally stored line number is set to 151 and the file name is changed to `copy.c`.

In the second example, the macro `ASSERT` uses the predefined identifiers **__LINE__** and **__FILE__** to print an error message about the source file if a given "assertion" is not true. Note that no quotation marks are necessary around **__FILE__**.

8.6 Pragmas

Syntax

pragma *character-sequence*

A #**pragma** is an implementation-defined instruction to the compiler. It has the general form given above, where *character-sequence* is a series of characters giving a specific compiler instruction and arguments, if any. The number sign (#) must be the first non-white-space character on the line containing the pragma; white-space characters can appear between the number sign and the word **pragma**.

See your system documentation for information about the pragmas available in your compiler implementation.

Language Reference Appendixes

Appendix A
Differences

This appendix summarizes differences between Microsoft C and the description of the C language found in Appendix A of *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, published in 1978 by Prentice-Hall, Inc. The following is a list of the differences, with cross-references to the corresponding section numbers in *The C Programming Language*:

Section Number in Kernighan and Ritchie	Microsoft C
2.2	Identifiers (including those used in preprocessor directives) are significant to 31 characters. External identifiers are also significant to 31 characters.
2.3	The identifiers **asm** and **entry** are no longer keywords. New keywords are **const**, **volatile**, **enum**, **signed**, and **void**. (The **const** and **volatile** keywords are not yet implemented but are reserved for future use.) The identifiers **cdecl**, **far**, **fortran**, **huge**, **near**, and **pascal** may be keywords, depending on whether or not the corresponding options are enabled when a program is compiled (see your system documentation).
2.4.1	As a result of the method used for assigning types to hexadecimal and octal constants, these constants always act like unsigned integers in type conversions.

2.4.3	Hexadecimal bit patterns consisting of a backslash (\), the letter **x**, and up to two hexadecimal digits are permitted as character constants (for example, `\x12`). Microsoft C defines three additional escape sequences: **v** represents a vertical tab (VT), \" represents the double-quote character, and **a** represents the bell (also called alert). Character constants always have type **int**, with the result that they are sign extended in type conversions.
2.6	The **short** type is always 16 bits in length, the **long** type 32 bits. The size of an **int** is machine dependent. On 8086/8088, 80186, and 80286 processors an **int** is 16 bits long, and on 80386 and 68000 machines it is 32 bits.
4	The **char** type is signed by default, with the result that a **char** value is sign extended in type conversions. (In some implementations, the default for the **char** type can be changed to unsigned at compile time.) Two additional unsigned types are supported: **unsigned char** and **unsigned long**. The keyword **unsigned** or **signed** can be applied as an adjective to an integer type. When **unsigned** appears alone it means **unsigned int**; similarly, when **signed** appears alone, it means **int**. Microsoft C offers an additional fundamental type, the **enum** (enumeration) type. The **void** type is defined as the return type of functions that do not return a value.

6.4	If the **near**, **far**, and **huge** keywords are enabled, pointers of different sizes may occur in a program. Operations with pointers of different sizes may cause conversion of pointers; the path of the conversion is implementation dependent.
6.6	The arithmetic conversions carried out by the Microsoft C Compiler are outlined in Sections 5.3.1 and 5.7 of Chapter 5, "Expressions and Assignments." Although compatible with the Kernighan and Ritchie conversions, the Microsoft C conversions are described in greater detail, including the specific path for each type of conversion. In addition to the usual arithmetic conversions, conversions between pointers of different sizes may be routinely carried out when the **near**, **far**, and **huge** keywords are enabled. The path of the pointer conversions is implementation dependent.
7.2	In connection with the **sizeof** operator, a byte is defined as an 8-bit quantity.
7.14	A structure can be assigned to another structure of the same type.
8.2	The keywords **enum** and **void** are additional type specifiers. The keyword **signed** or **unsigned** can serve either as a type specifier or as an adjective modifying an integer type.

	Therefore, the following additional combinations are acceptable: **signed char** **signed short** **signed short int** **signed long** **signed long int** **unsigned char** **unsigned short** **unsigned short int** **unsigned long** **unsigned long int**
8.4	Optional argument-type lists can be included in function declarations to notify the compiler of the number and types of arguments expected in a function call.
8.5	Bit fields must be declared **unsigned**. The names of structure and union members are not required to be distinct from structure and union tags or from the names of other variables. No relationship exists between the members of two different structure types.
8.6	Unions can be initialized by giving a value for the first member of the union.
9.7	The *expression* of a **switch** is an integral expression that is the size of an **int** or shorter. An *expression* with **enum** type is permitted. Each of the **case** constant expressions is cast to the type of the *expression*.
10.1	The parameter list in a function definition can end with a comma followed by three periods (,...) or just a comma (,) to indicate that the number of parameters is variable. A

parameter list containing only three periods (...) or a comma (,) indicates that the function can take zero or more parameters.

12

The number sign (#) introducing the preprocessor directive can be preceded by any combination of white-space characters. White space can also occur between the number sign and the preprocessor keyword.

In addition to preprocessor directives, the source file can also contain pragmas. Pragmas, like directives, are introduced by a number sign as the first non-white-space character in a line. The action defined by a particular pragma is implementation dependent.

12.3

The new combination #**if** **defined**(*identifier*) is intended to supplant the #**ifdef** and #**ifndef** directives. Use of the latter directives is discouraged.

The new directive #**elif** (else-if) is designed for use in #**if** and #**if** **defined** blocks.

14.1

A structure or union can be assigned to another structure or union of the same type. Structures and unions can be passed by value to functions and returned by functions.

In expressions involving $->$, the expression preceding the arrow must have the same type (or be cast to the same type) as the structure to which the member on the right-hand side of the arrow belongs.

17

The listed anachronisms are not recognized.

Appendix B
Syntax Summary

B.1 Tokens

keyword
identifier
constant
string
operator
separator

B.1.1 Keywords

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const*	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile***
do	**if**	**static**	**while**

* Not yet implemented

The following identifiers may be keywords in some implementations. See your system documentation for information.

cdecl
far
fortran
huge
near
pascal

B.1.2 Identifiers

identifier:
    letter
    underscore
    identifier letter
    identifier underscore
    identifier digit

letter–one of the following:
a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

underscore:
_

digit–one of the following:
0 1 2 3 4 5 6 7 8 9

B.1.3 Constants

constant:
integer-constant
long-constant
floating-point-constant
char-constant
enum-constant

integer-constant:
0
decimal-constant
octal-constant
hexadecimal-constant

decimal-constant:
nonzero-digit
decimal-constant digit

nonzero-digit–one of the following:
1 2 3 4 5 6 7 8 9

octal-constant:
0*octal-digit*
octal-constant octal-digit

octal-digit–one of the following:
0 1 2 3 4 5 6 7

hexadecimal-constant:
0x*hexadecimal-digit*
0X*hexadecimal-digit*
hexadecimal-constant hexadecimal-digit

hexadecimal-digit–one of the following:
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

long-constant:
integer-constant l
integer-constant L

floating-point-constant:
fractional-constant exponent
fractional-constant
digit-seq exponent

fractional-constant:
digit-seq . *digit-seq*
. *digit-seq*
digit-seq .

digit-seq:
digit
digit-seq digit

exponent:
e *sign digit-seq*
E *sign digit-seq*
e *digit-seq*
E *digit-seq*

sign:
+
–

char-constant:
'*char*'

char:
rep-char
escape-sequence

rep-char:
Any single representable character except the single quote (’), backslash (\), or new-line character

escape-sequence–one of the following:

\’	\"	\\	*ddd*	\x*ddd*	\a
\b	\f	\n	\r	\t	\v

enum-constant:
identifier

B.1.4 Strings

string-literal:
""
"*char-seq*"

char-seq:
char
char-seq char

B.1.5 Operators

operator–one of the following:

!	~	++	--	+
-	*	/	%	<<
>>	<	<=	>	>=
==	!=	|	&	^
&&	||	=	+=	-=
*=	/=	%=	>>=	<<=
&=	^=	|=	?:	,
[]	()	.	->	

B.1.6 Separators

separator–one of the following:

```
[   ]   (   )       {   }
*   ,   :   =       ;   #
```

B.2 Expressions

expression:
- *identifier*
- *constant*
- *string*
- *expression*(*expression-list*)
- *expression*( )
- *expression*[*expression*]
- *expression*.*identifier*
- *expression*–>*identifier*
- *unary-expression*
- *binary-expression*
- *ternary-expression*
- *assignment-expression*
- (*expression*)
- (*type-name*)*expression*
- *constant-expression*

expression-list:
- *expression*
- *expression-list* , *expression*

unary-expression:
- *unop expression*
- **sizeof**(*expression*)

unop–one of the following:

```
-   ~   !   *   &
```

lvalue:
- *identifier*
- *expression*[*expression*]
- *expression*.*expression*
- *expression*–>*expression*
- **expression*

(type-name)expression
(lvalue)

type-name:
See Section B.3, "Declarations."

binary-expression:
expression binop expression

binop–one of the following:

*	/	%	+	-
<<	>>	<	>	<=
>=	==	!=	&	|
^	&&	||	,	

ternary-expression:
expression ? *expression* : *expression*

assignment-expression:
lvalue++
lvalue--
++*lvalue*
--*lvalue*
lvalue assignment-op expression

assignment-op–one of the following:

=	*=	/=	%=	+=	-=
<<=	>>=	&=	|=	^=	

constant-expression:
identifier
constant
(type-name)constant-expression
unary-expression
binary-expression
ternary-expression
(constant-expression)

B.3 Declarations

declaration:
    sc-specifier type-specifier declarator-list;
    type-specifier declarator-list;
    sc-specifier declarator-list;
    type-specifier;
    typedef *type-specifier declarator-list*;

sc-specifier:
    auto
    extern
    register
    static

type-specifier:
    char
    double
    enum-specifier
    float
    int
    long
    long int
    short
    short int
    struct-specifier
    typedef-name
    union-specifier
    unsigned
    unsigned char
    unsigned int
    unsigned long
    unsigned long int
    unsigned short
    unsigned short int
    signed
    signed char
    signed int
    signed long
    signed long int
    signed short
    signed short int

enum-specifier:
    enum *tag* { *enum-list*}

enum { *enum-list* }
enum *tag*

tag:
identifier

enum-list:
enumerator
enum-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

struct-specifier:
struct *tag* { *member-declaration-list* }
struct { *member-declaration-list* }
struct *tag*

member-declaration-list:
member-declaration
member-declaration-list member-declaration

member-declaration:
type-specifier declarator-list;
type-specifier identifier : *constant-expression*;
type-specifier : *constant-expression*;

declarator-list:
declarator
declarator = *initializer*
declarator-list , *declarator*

declarator:
identifier
range-modifier identifier
declarator[ ]
declarator[*constant-expression*]
**declarator*
declarator( )
declarator(*arg-type-list*)
(*declarator*)

arg-type-list:
type-name
arg-type-list, *type-name*

arg-type-list,...
arg-type-list,
void
void *
...
,

type-name:
type-specifier
type-specifier abstract-declarator

abstract-declarator:
*
*range-modifier**
[ ]
(*arg-type-list*)
**abstract-declarator*
*abstract-declarator**
abstract-declarator[ ]
abstract-declarator[*constant-expression*]
[ ]*abstract-declarator*
[*constant-expression*]*abstract-declarator*
abstract-declarator( )
abstract-declarator(*arg-type-list*)
(*abstract-declarator*)

initializer:
expression
{ *initializer-list*}

initializer-list:
initializer
initializer-list, *initializer*

typedef-name:
identifier

union-specifier:
union *tag* { *member-declaration-list*}
union { *member-declaration-list*}
union *tag*

range-modifier:
cdecl
far
fortran

huge
near
pascal

B.4 Statements

statement:
break;
case *constant-expression* **:** *statement*
compound-statement
continue;
default : *statement*
do *statement* **while(***expression***);**
*expression***;**
for ([[*expression*]]**;**[[*expression*]]**;**[[*expression*]]**)** *statement***;**
goto *identifier***;**
identifier **:** *statement*
if (*expression***)** *statement* [[**else** *statement*]]
;
return [[*expression*]]**;**
switch (*expression***)** *statement*
while (*expression***)** *statement*

compound-statement:
{ [[*declaration-list*]] [[*statement-list*]] **}**

declaration-list:
declaration
declaration-list declaration

statement-list:
statement
statement-list statement

B.5 Definitions

definition:
    function-definition
    data-definition

function-definition:
    [[*sc-specifier*]] [[*type-specifier*]] *declarator* ([[*parameter-list*]]) [[*parameter-decs*]] *compound-statement*

parameter-list:
    fixed-parameter-list
    variable-parameter-list

fixed-parameter-list:
    identifier
    parameter-list , *identifier*

variable-parameter-list:
    fixed-parameter-list,...
    fixed-parameter-list,
    ...
    ,

parameter-decs:
    declaration
    declaration-list declaration

data-definition:
    declaration

B.6 Preprocessor Directives

directive:
    #
    #**define** *identifier* [[([[*parameter-list*]])]] [[*token-seq*]]
    #**elif** *restricted-constant-expression*
    #**else**
    #**endif**
    #**if** *restricted-constant-expression*
    #**ifdef** *identifier*
    #**ifndef** *identifier*

#**include** "*string*"
#**include** <*string*>
#**line** *digit-seq*
#**line** *digit-seq string*
#**undef** *identifier*

token-seq:
token
token-seq token

restricted-constant-expression:
defined (*identifier*)
Any *constant-expression* except for **sizeof** expressions, casts, and enumeration constants

B.7 Pragmas

pragma:
#**pragma** *char-seq*

Language Reference Index

MICROSOFT®
16011 NE 36th Way, Box 97017, Redmond, WA 98073-9717

Software Problem Report

Name _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Instructions

Use this form to report software bugs, documentation errors, or suggested enhancements. Mail the form to Microsoft.

Category

_____ Software Problem

_____ Software Enhancement

_____ Documentation Problem
(Document #_____)

_____ Other

Software Description

Microsoft Product _____

Rev. _____ Registration # _____

Operating System _____

Rev. _____ Supplier _____

Other Software Used _____

Rev. _____ Supplier _____

Hardware Description

Manufacturer _____ CPU _____ Memory _____ KB

Disk Size _____" Density: Sides:

Single _____ Single _____

Double _____ Double _____

Peripherals _____

Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

Microsoft Use Only

Tech Support _____ Date Received _____

Routing Code _____ Date Resolved _____

Report Number _____

Action Taken:

MICROSOFT®